



UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA

*Departamento de Electrónica y Sistemas*

Proxecto de Fin de Carreira de Enxeñería Técnica en Informática  
de Sistemas

**Programación de comportamientos básicos para  
el robot humanoide Nao: tirar penaltis**

*Autor:* Héctor Gómez Varela

*Tutor:* Carlos Vázquez Regueiro

A Coruña, 17 de septiembre de 2014



# Especificación

*Título:* Programación de comportamientos básicos para el robot humanoide Nao.

*Clase:* Proyecto clásico de ingeniería.

*Autor:* Héctor Gómez Varela.

*Director:* Carlos Vázquez Regueiro.

*Tribunal:* Carlos Vázquez Regueiro.

*Fecha de lectura:* 15/09/2014

*Calificación:*



*A Arene, por su ayuda, cariño y comprensión.*

*A mi familia, por su apoyo durante toda la carrera, en los buenos y malos momentos.*



# Agradecimientos

Agradezco muy especialmente a mi tutor, Carlos Vázquez Regueiro, todas las horas de guía y toda la paciencia.

Al CITIUS (Centro Singular de Investigación en Tecnoloxías da Información) de la USC (Universidade de Santiago de Compostela), y en especial al profesor Roberto Iglesias tanto la cesión del robot Nao para efectuar las pruebas experimentales, como la licencia del simulador Webots, sin el que el diseño de este software hubiera resultado imposible.

A Luis Calvo, por toda la ayuda prestada y por hacerme de guía en varias ocasiones.



# **Resumen**

Se ha diseñado e implementado un sistema de control y respuesta al entorno para un robot humanoide Nao, fabricado por Aldebaran Robotics. Siguiendo una metodología ágil, se dota al robot de la capacidad de percibir una pelota en el entorno, acercarse a ella controladamente, y ejercer un golpeo también controlado y parametrizable con una de sus piernas. Se ha usado el propio software del fabricante, y se ha establecido una arquitectura que descansa sobre el sistema operativo robótico ROS, implementada como un conjunto de nodos independientes escritos en Python.

Se ha dispuesto, además, de un robot Nao real, con el que se ha podido probar los algoritmos diseñados, lo que ha dado lugar a la documentación del proyecto mediante pruebas experimentales.

## **Palabras clave**

Robotica, Nao, Webots, ROS, Python, visión artificial, robocup, penaltis.



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Contexto . . . . .	1
1.2. Objetivos . . . . .	1
1.3. Estructura de la memoria . . . . .	2
<b>2. Fundamentos teóricos</b>	<b>5</b>
2.1. Robótica . . . . .	5
2.1.1. Qué es la robótica . . . . .	5
2.1.2. Historia de la robótica . . . . .	6
2.2. El Robot NAO . . . . .	10
2.2.1. Movimiento . . . . .	11
2.2.2. Hardware del Nao . . . . .	12
2.2.3. Software del Nao . . . . .	15
2.3. Robot Operating System (ROS) . . . . .	20
2.3.1. Sistema de ficheros de ROS . . . . .	20
2.3.2. Sistema de comunicación y computación de ROS . . . . .	21
2.4. Estado del arte . . . . .	24
2.4.1. Diseño del comportamiento de un portero de fútbol robótico . .	25
2.4.2. Construcción y pruebas en Gazebo y V-Rep de un modelo para el robot humanoide Nao . . . . .	25
2.4.3. Aprendizaje por Refuerzo Seguro para enseñar a un robot humanoide a caminar más rápido . . . . .	26
2.4.4. Robocup . . . . .	26

<b>3. Valoración de hardware y software</b>	<b>29</b>
3.1. Tecnologías utilizadas . . . . .	29
3.2. Sistema operativo local . . . . .	31
3.3. Sistema operativo de gestión del robot (ROS) . . . . .	31
3.3.1. Justificación de uso de ROS . . . . .	31
3.3.2. Justificación del uso de Python para implementar nodos ROS . .	33
3.3.3. Conclusiones . . . . .	34
3.4. Sistemas de representación de objetos 3D . . . . .	34
3.4.1. Unified Robot Description Format (URDF) . . . . .	35
3.4.2. Simulation Description Format (SDF) . . . . .	36
3.5. Simuladores . . . . .	37
3.5.1. Gazebo . . . . .	38
3.5.2. V-REP . . . . .	41
3.5.3. Webots . . . . .	43
3.5.4. Conclusiones . . . . .	43
3.6. Alternativas dentro de Naoqi . . . . .	45
3.6.1. Alternativas para el procesamiento de imágenes . . . . .	45
3.6.2. Ejecución de movimientos . . . . .	49
<b>4. Metodología</b>	<b>55</b>
4.1. Metodología ágil iterativa-incremental . . . . .	55
4.2. Fase de análisis inicial . . . . .	56
4.3. Fase de estudio de la tecnología . . . . .	56
4.4. Iteraciones . . . . .	57
4.4.1. Fase de análisis . . . . .	57
4.4.2. Fase de diseño . . . . .	57
4.4.3. Fase de implementación . . . . .	58
4.4.4. Fase de prueba . . . . .	58
<b>5. Análisis</b>	<b>61</b>
5.1. Análisis de requisitos básicos . . . . .	61

5.2. Análisis de tareas funcionales . . . . .	62
5.3. Objetivos concretos del trabajo . . . . .	63
<b>6. Planificación y costes</b>	<b>65</b>
6.1. Planificación . . . . .	65
6.1.1. Estudio de la tecnología . . . . .	66
6.1.2. Fase de análisis inicial . . . . .	66
6.1.3. Iteraciones funcionales y pruebas . . . . .	66
6.1.4. Pruebas con el robot real . . . . .	67
6.1.5. Diagrama de planificación . . . . .	67
6.2. Costes . . . . .	68
6.2.1. Recursos humanos . . . . .	68
6.2.2. Hardware y software . . . . .	68
6.2.3. Hardware . . . . .	69
6.3. Resultados de la planificación . . . . .	70
<b>7. Diseño</b>	<b>71</b>
7.1. Diseño de funcionalidades . . . . .	71
7.2. Arquitectura global del software . . . . .	72
7.2.1. Detección de la pelota . . . . .	74
7.2.2. Desplazamiento estable . . . . .	74
7.2.3. Desplazamiento adaptativo . . . . .	76
7.2.4. Golpeo estable . . . . .	77
7.2.5. Golpeo configurable . . . . .	78
<b>8. Implementación</b>	<b>81</b>
8.1. Módulo de inicialización (init_robot) . . . . .	82
8.2. Módulo de gestión de las cámaras (camera) . . . . .	82
8.3. Módulo detección (ball_pos) . . . . .	82
8.4. Módulo de desplazamiento (walker) . . . . .	84
8.5. Módulo de ajuste de posición para golpeo (set_position) . . . . .	85
8.6. Módulo de golpeo (kicker) . . . . .	86

<b>9. Pruebas experimentales</b>	<b>89</b>
9.1. Golpeo con ángulo variable . . . . .	89
9.1.1. Objetivo de la prueba . . . . .	90
9.1.2. Resultados en simulación . . . . .	90
9.1.3. Resultados con robot real . . . . .	90
9.1.4. Discusión . . . . .	95
9.2. Detección de pelota . . . . .	95
9.2.1. Objetivo de la prueba . . . . .	95
9.2.2. Resultados en simulación . . . . .	96
9.2.3. Discusión . . . . .	99
9.3. Detección de pelota y posicionamiento de robot . . . . .	99
9.3.1. Objetivo de la prueba . . . . .	99
9.3.2. Resultados en simulación . . . . .	100
9.3.3. Resultados con robot real . . . . .	100
9.3.4. Discusión . . . . .	101
<b>10. Conclusiones y trabajo futuro</b>	<b>103</b>
10.1. Conclusiones . . . . .	103
10.2. Trabajo futuro . . . . .	104
<b>A. Protocolo de comunicaciones XML-RPC</b>	<b>107</b>
A.1. Introducción a XML-RPC . . . . .	107
<b>B. Manual de usuario</b>	<b>109</b>
<b>C. Contenido del DVD</b>	<b>111</b>
<b>D. Referencias bibliográficas</b>	<b>113</b>

# Índice de figuras

2.1.	Eolípila de Herón. . . . .	6
2.2.	Autómata de Leonardo Da Vinci reconstruido. . . . .	7
2.3.	Telar automatizado diseñado y construido por Jacquard . . . . .	7
2.4.	Robot industrial Unimate (1956) . . . . .	8
2.5.	Shakey, primer autómata móvil autónomo (1962) . . . . .	9
2.6.	Lunokhod 1, primer robot lunar teledirigido . . . . .	9
2.7.	Robot ASIMO, desarrollado por Honda (2000) . . . . .	10
2.8.	Robot Nao, vista frontal y cenital . . . . .	11
2.9.	Diversas articulaciones del robot humanoide Nao. . . . .	12
2.10.	Posición de los sensores de ultrasonidos, situados en el torso del Nao. . . . .	13
2.11.	Posición y ángulos efectivos de las cámaras del Nao . . . . .	14
2.12.	Choregraphe en funcionamiento. . . . .	16
2.13.	Diagrama jerárquico de NaoQi . . . . .	17
2.14.	Llamada a método con bloqueo . . . . .	18
2.15.	Llamada a método sin bloqueo o en paralelo . . . . .	18
2.16.	Sistema de comunicación ROS sencillo . . . . .	23
2.17.	Nao en el terreno de juego de la Robocup. . . . .	27
3.1.	Conjunto de paquetes ROS que proporcionan la interpretación de URDF	35
3.2.	Diagrama del proceso de adecuación del modelo de Nao al simulador Gazebo . . . . .	40
3.3.	Interfaz gráfica y simulación en V-REP . . . . .	42
3.4.	Interfaz gráfica y simulación mediante Webots . . . . .	44

4.1.	Flujo de fases presentes en la metodología seguida . . . . .	56
5.1.	Estructura de desarrollo del comportamiento final. . . . .	63
5.2.	Flujo de fases en la creación de una funcionalidad. . . . .	64
6.1.	Diagrama de Gantt aproximado sobre la gestión del proyecto. . . . .	67
7.1.	Arquitectura global del software. . . . .	73
7.2.	Diseño del módulo de detección de posiciones de la pelota. . . . .	75
7.3.	Diseño del módulo de desplazamiento estable con trayectoria variable. .	75
7.4.	Diseño del módulo de desplazamiento adaptativo. . . . .	77
7.5.	Diseño del módulo de ajuste posicional. . . . .	78
7.6.	Diseño del módulo de golpeo básico. . . . .	79
7.7.	Diseño del módulo de golpeo configurable o adaptable. . . . .	80
8.1.	Sistema de referencia definido por Naoqi para la implementación de movimientos coordinados. . . . .	83
8.2.	Nao en el instante previo al golpeo con pierna derecha. . . . .	88
8.3.	Vista cenital de la trayectoria a través del eje Y. . . . .	88
8.4.	Vista lateral de la trayectoria a través del eje Z. . . . .	88
9.1.	Captura de la propia cámara del robot, detectando la bola. . . . .	92
9.2.	Captura de cámara externa, con Nao golpeando la pelota durante el experimento. . . . .	93
9.3.	Patada modificada respecto al eje Y para el ajuste del golpeo. . . . .	93
9.4.	Patada modificada respecto a los ejes Y,Z para el ajuste del golpeo. .	94
9.5.	Gráfico de precisión del nodo ball_pos para desplazamiento de la bola a través del eje X. . . . .	98
9.6.	Gráfico de precisión del nodo ball_pos para desplazamiento de la bola a través del eje Y. . . . .	99

# **Capítulo 1**

## **Introducción**

### **1.1. Contexto**

Uno de los grandes anhelos humanos a lo largo de los siglos ha sido el de delegar en los robots la carga de trabajos pesados, peligrosos o tediosos. Hemos querido automatizar estas tareas, y es por ello que hay cierta carga de utilitarismo en el progreso de la robótica. Pero también hay fascinación porque aquellos autómatas de la antigüedad puedan convertirse hoy en máquinas que se parecen a nosotros y gozan de cierta capacidad de acomodación a su entorno. Máquinas autónomas gobernadas por una inteligencia artificial en exponencial desarrollo.

### **1.2. Objetivos**

En una primera aproximación, se propone como objetivo el diseño y programación de comportamientos sencillos para el robot humanoide Nao. Los comportamientos que se tuvieron en cuenta buscaban la interacción de diferentes partes clave en la programación de máquinas autónomas. Entre los comportamientos y funciones consideradas se incluye por ejemplo la detección de cuerpos en movimiento, la respuesta del robot a estos estímulos entre otros, o la capacidad de realizar movimientos con cierto nivel de complejidad, siempre de forma adaptable a las condiciones del entorno. Se propone

también una metodología ágil iterativa, abierta a modificaciones durante el desarrollo, de tal forma que los objetivos del proyecto puedan variar en función de las consecuencias de objetivos parciales.

Más en concreto, los objetivos del trabajo, en cuya conjunción podríamos ver reflejado el objetivo final del proyecto, es decir, que el Nao tire un penalti, y que se definieron en etapas más avanzadas de la planificación, los exponemos aquí:

1. **Desarrollo de un golpeo estable**, que no genere inestabilidad en el robot.
2. **Detección de la pelota en el entorno** de forma que el sistema pueda proporcionar en cada instante de tiempo la posición de ésta.
3. **Diseño de un algoritmo de acercamiento y posicionamiento** del robot con respecto a la pelota, situándose de forma que puede ejercerse el golpeo.
4. **Golpeo variable** de la pelota, de tal forma que podamos orientar la trayectoria de ésta dependiendo de la patada ejecutada.
5. **Detección de un objeto externo**, que simbolizaría una portería y sería el objetivo que tendría que alcanzar la pelota tras ser golpeada por el robot.

### 1.3. Estructura de la memoria

1. **Introducción**: se contextualiza y se expone la motivación para la realización del proyecto. Se expone la estructura de éste.
2. **Fundamentos teóricos**: en este capítulo se hace un repaso histórico por la robótica, así como por las aplicaciones de ésta. Se exponen las tecnologías básicas con las que se trabajará y se expone una justificación de su uso.
3. **Valoración de hardware y software**: se justifica el uso de ROS y del propio software del fabricante del robot empleado, Naoqi. También se hace un repaso por las tecnologías que se han barajado para dar soporte al problema, tanto las que se han elegido finalmente como las que no, y las motivaciones de estas decisiones.

4. **Metodología:** en este capítulo se explica la metodología seguida, así como el porqué de seguir ésta y no otra.
5. **Análisis:** se explica la fase de análisis inicial del proyecto, y, basándose en las capacidades del hardware y software que se ha elegido, se establecen una serie de tareas funcionales que se pretenden cubrir durante el desarrollo.
6. **Planificación y costes:** se expone el uso del tiempo efectivo para el proyecto, su repartición en diferentes fases de trabajo. Se exponen también los costes monetarios de estas fases.
7. **Diseño:** en este capítulo se muestra tanto la arquitectura completa del software final, como el diseño particularizado para cada una de las entidades lógicas que conforman la arquitectura global.
8. **Implementación:** siguiendo el diseño lógico del software que se ha presentado en el capítulo 7, se comenta la implementación en módulos ROS-Python que la sustenta.
9. **Pruebas experimentales:** se exponen las diferentes pruebas que se han realizado sobre el software, orientando éstas a la consecución de diferentes hitos de desarrollo.
10. **Apéndice A, Protocolo de comunicaciones XML-RPC:** se comenta a modo documental el protocolo de comunicaciones sobre el que se construye el transporte de ROS, XML-RPC.
11. **Apéndice B, Manual de usuario:** se proporcionan las líneas básicas para la instalación y puesta en funcionamiento del software que se ha diseñado.
12. **Apéndice C, Contenido del DVD:** se proporciona una lista y una explicación concisa del contenido del DVD que se adjunta con este trabajo.



# Capítulo 2

## Fundamentos teóricos

En este capítulo se hará un breve recorrido por la historia de la robótica a lo largo de la historia, y por sus hitos más destacados. También se expondrá el software básico sobre el que desde un principio se ha trabajado: ROS y Naoqi. Se hará también una introducción de algunos de los trabajos que se sitúan en el mismo campo que el presente, y hablaremos de la Robocup, competición que enfrenta a equipos de robots y se ha tomado como inspiración y motivación para el proyecto.

### 2.1. Robótica

#### 2.1.1. Qué es la robótica

La robótica es la ciencia que se dedica al diseño, construcción, implementación y aplicación de estos ingenios electrónicos programables. La robótica es una ciencia con miles de años de historia[10], pero sólo a partir de 1921 conocemos a éstos con el nombre de *robots*, después de que el escritor checo Karel Čapek(1890–1938) acuñase el término en su obra de teatro con tintes de ciencia ficción *Rossumovi Univerzální Roboti -Robots Universales Rossum*[11]. Es por ello que *robot* es una palabra cuyo origen es puramente checo.

Podemos ver en la robótica un influjo de otras ciencias, tales como la electrónica y la mecánica. Pero el papel de la informática y la inteligencia artifical ha cobrado gran

importancia con los tiempos.

### 2.1.2. Historia de la robótica

Existen autómatas desde incluso varios siglos antes de Cristo, como por ejemplo los relojes de agua que se construyeron para el rey Amenhotep I en Egipto, aproximadamente en el año 1530 a.C. Pero no sería hasta los trabajos del griego Herón de Alejandría (10-70 d. C.), ingeniero y matemático, que se implementaron lo que podríamos llamar en máquinas en el amplio sentido de la palabra. Herón de Alejandría diseñó ingenios mecánicos que eran capaces de describir movimientos repetitivos y orientados a algún fin, por sí mismos, y únicamente mediante la adquisición de energía proveniente de algún agente externo.

Entre estos progresos, destaca por ejemplo su eolípila, que transformaba la energía calorífica del fuego en movimiento jugando con el vapor y la presión que el agua ejercía sobre sus conductos.



Figura 2.1: Eolípila de Herón.

Unos cuantos siglos después, concretamente en el año 1495 d.C., Leonardo Da Vinci (1452–1519) diseña el primer robot humanoide, aunque no está claro que durante la vida de Leonardo el robot llegase a implementarse realmente. Sí se ha construido en varias ocasiones en la contemporaneidad, demostrando así que el diseño de Leonardo podía agitar los brazos, incorporarse, mover el cuello e incluso la mandíbula.

Sin duda la Revolución Industrial jugó un papel decisivo en la progresión de la robótica. El ser humano percibió en esta etapa la posibilidad de finalmente ir susti-

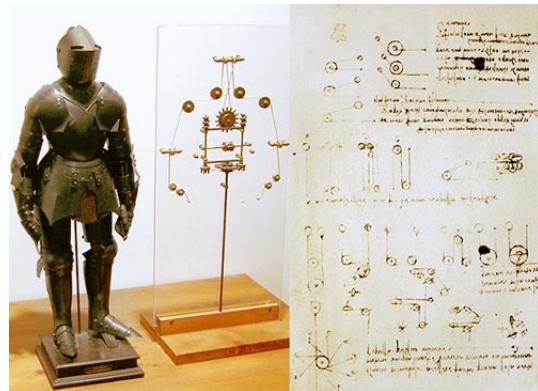


Figura 2.2: Autómata de Leonardo Da Vinci reconstruido.

tuyendo a las personas por autómatas en cuanto a la realización de las tareas más peligrosas, pesadas o tediosas. Así, las máquinas se especializaron en aquellas tareas que exigían un alto grado de repetición, y -en muchos casos- necesitaban ser mínimamente gobernadas por un patrón de trabajo externo. En este sentido, destacan por lo que en el futuro representarían, los telares programables de Joseph Marie Jacquard (1752–1834), que usaban tarjetas de cartón perforadas para la gestión del trabajo, y que sirvieron como paradigma de las primeras máquinas computacionales de la historia.

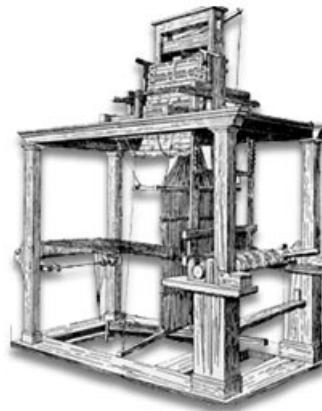


Figura 2.3: Telar automatizado diseñado y construido por Jacquard

Ya en el siglo XX, a finales de la década de los años 30, se diseñan los primeros brazos articulados, principalmente para la realización de trabajos de pintura y chapa

en fábricas de producción de aparatos eléctricos. La primera implementación conjunta de un brazo articulado y un computador se produjo en el año 1956. George Devol y Joseph Engelberger construyeron un robot llamado Unimate, que podía tomar pequeñas decisiones de forma autónoma para realizar tareas de forma automática. Fue instalado, en 1962, en la planta de General Motors de New Jersey, convirtiéndose en el primer robot industrial.



Figura 2.4: Robot industrial Unimate (1956)

El primer autómata móvil capaz de percibir su entorno y reaccionar de forma acorde a las condiciones de éste fue Shakey[12], cuyo diseño fue desarrollado entre 1966 y 1972 con Charles Rosen como director de proyecto. Shakey no necesitaba ser instruído en cada movimiento que realizaba, sino que podía realizar tareas complejas, acomodando pasos más sencillos a su consecución. Fue diseñado y construido en el Artificial Intelligence Center of Stanford, y fue el primer robot que incorporó técnicas de visión artificial y procesamiento del lenguaje natural a su desarrollo. Poseía una cámara y diversos sensores para medir distancias y evitar colisiones.

La carrera espacial también se ha beneficiado y a invertido esfuerzos en el campo de la robótica. Se ha visto que los robots son una clara alternativa a los humanos en la exploración de entornos que se hacen difícilmente habitables para los seres humanos. En este aspecto, cabe señalar como primer uso de la tecnología robótica el Lunokhod 1[13], autómata que uso la Unión Soviética para la exploración lunar en 1970, y que



Figura 2.5: Shakey, primer autómata móvil autónomo (1962)

por consiguiente constituyó el primer autómata controlado de forma remota.

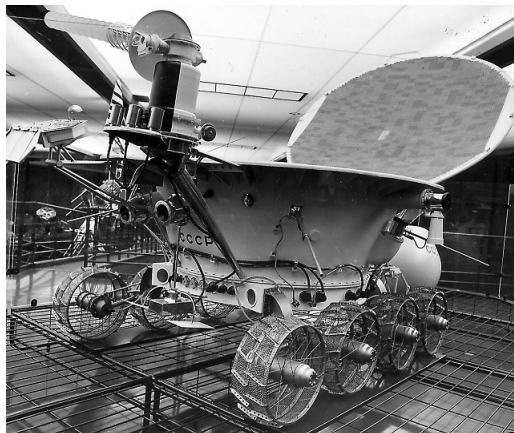


Figura 2.6: Lunokhod 1, primer robot lunar teledirigido

En el año 2000, la compañía japonesa Honda presentó su robot ASIMO[14], un proyecto que surgió con la intención de crear un autómata que pudiese emular o imitar con el mayor grado de aproximación posible el andar humano. Para ello, se ha basado su diseño en la posición y la proporción de las articulaciones del cuerpo hu-

mano. Asimismo, ASIMO incorpora mecanismos de reconocimiento visual y sonoro, y es capaz de tomar decisiones complejas basadas en sus propias percepciones del entorno.

En la actualidad la robótica se ha diversificado en cierta medida, un ejemplo de aplicación a otras disciplinas, o de simbiosis tecnológica es por ejemplo lo que ha venido a llamarse nanorobótica, que trabaja con el diseño y construcción de robots a escala nanométrica ( $10e-9$  metros), y cuyos nanorobots intentan interactuar microorganismos celulares, realizando tareas que son particularmente útiles en los campos de la microcirugía y la biomédicina en general.



Figura 2.7: Robot ASIMO, desarrollado por Honda (2000)

Existen también otras formas de experimentación con máquinas autónomas, como pueden ser los robot reconfigurables, cuyo desarrollo comprende aquellos ingenios que idealmente podrían ser capaces de reconstruirse a sí mismos, modificando sus atributos físicos de tal forma que pudiesen adecuarse al entorno que los rodea. E incluso existen estudios que trabajan con robots flexibles, cuya anatomía podría asemejarse idealmente a la de los seres vivos.

## 2.2. El Robot NAO

Nao es un robot humanoide de 58 cm. Ha sido diseñado y construido por Aldebaran Robotics, compañía de origen francés con sede en París, que comenzó a trabajar en el

robot en 2004, aunque éste no sería distribuido hasta 2008. Ese año se pudieron adquirir los primeros robots. Tan pronto como estos primeros modelos vieron la luz, Nao se convirtió en el robot oficial de la Robocup (Robot Soccer World Cup), competición de carácter experimental que fomenta la investigación en el ámbito de la inteligencia articial y la cooperación entre sistemas robóticos en contextos dinámicos.

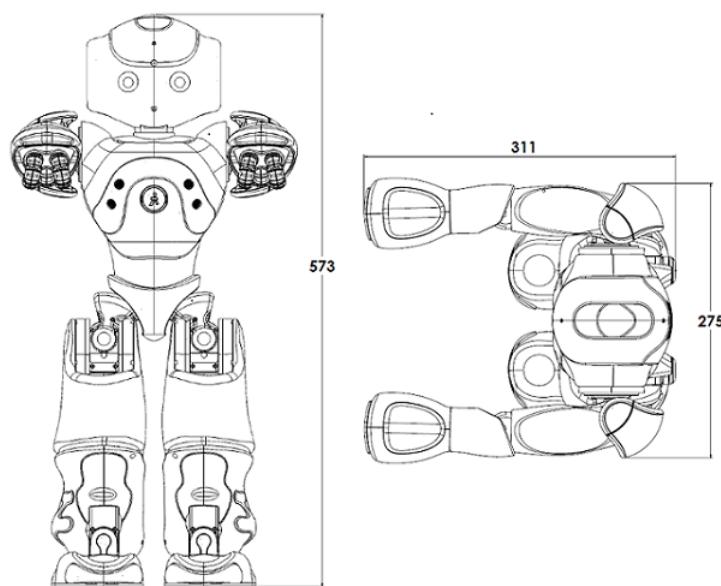


Figura 2.8: Robot Nao, vista frontal y cenital

Para la realización de este trabajo, hemos dispuesto de la versión 3.3 del robot. Esto hace que no podamos tener acceso a mejoras que sí incorporan algunas de las versiones más recientes del robot, como es la conexión USB, o la incorporación de un procesador más potente. En nuestra versión, nos vemos a trabajar con un AMD Geode, con una frecuencia de reloj de 550 Mhz, mientras que el Nao V4.0 incorpora ya Intel Atom a 1.6GHz. También disponemos de 256MB de memoria SDRAM, una cantidad más reducida que en la versión 4.0, asciende a 2GB.

### 2.2.1. Movimiento

El robot dispone de hasta 25 grados de libertad, que se corresponden con motores eléctricos y actuadores, dos de los cuales se encuentran situados en la cabeza, cuatro en

cada uno de los dos brazos, uno en cada una de las manos, y seis en cada pierna. Esta gran cantidad de actuadores permiten al robot tener una gran capacidad para realizar movimientos complejos, en los que intervienen una gran cantidad de articulaciones. Nao cuenta incluso con un sistema que permite al robot levantarse en el caso de que ocurra una caída al suelo durante la ejecución de algún movimiento. Muchas de las acciones que Nao realiza, en imitación de un comportamiento humano estándar, implican el trabajo conjunto de prácticamente todos los actuadores del robot.

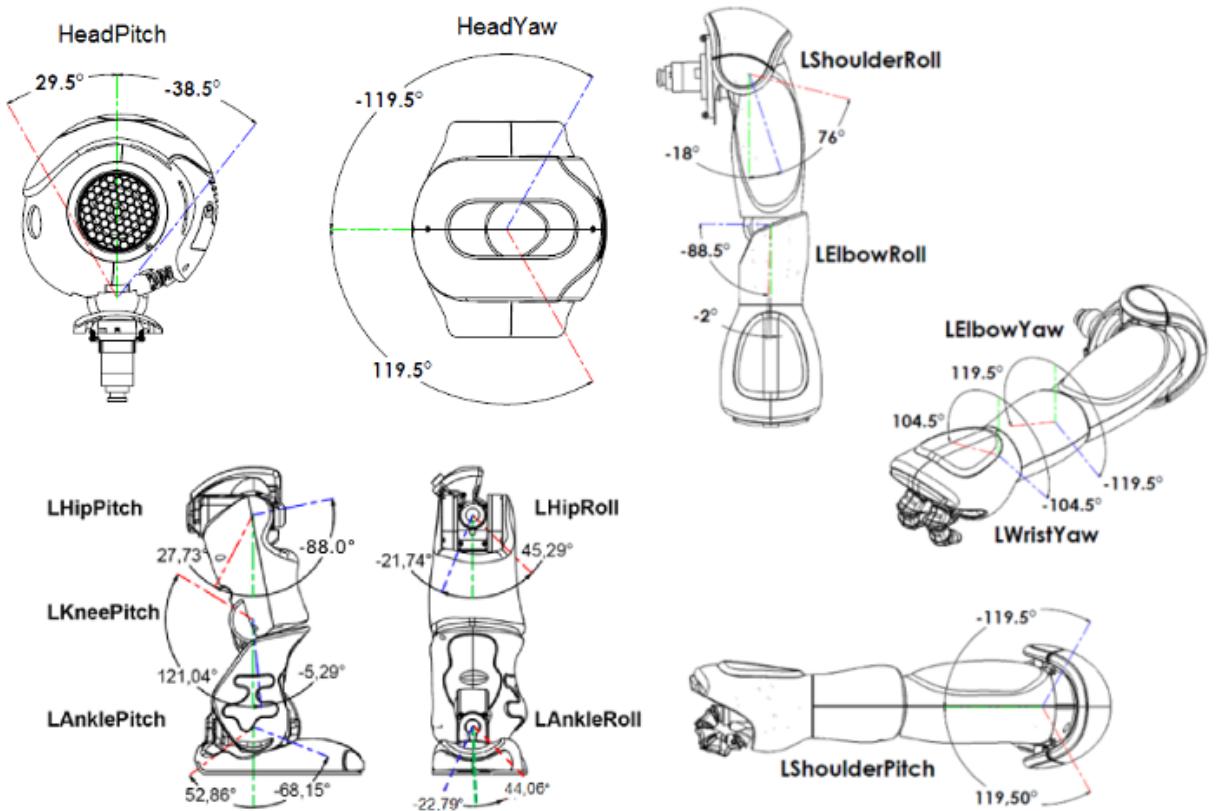


Figura 2.9: Diversas articulaciones del robot humanoide Nao.

### 2.2.2. Hardware del Nao

La relación con el entorno podría pensarse como un aspecto ineludible en la construcción de cualquier máquina autónoma. Para la toma de datos del mundo exterior,

Nao está equipado con 14 sensores táctiles, que se reparten entre cabeza, pies, manos y torso; unidos a dos emisores/receptores de ultrasonidos (sonars), un sistema inercial (formado por dos giroscopios y un acelerómetro), dos cámaras (ambas situadas en la cabeza del robot, con diferentes rangos de visión), dos emisores/receptores de señales infrarrojas, y varios altavoces y micrófonos.

### Sensores de ultrasonidos

Para el desarrollo de este proyecto, los dos sistemas de importancia crítica son los sensores de ultrasonidos y las dos cámaras que Nao incorpora. Los tanto los receptores como los emisores de ultrasonidos se encuentran situados en el torso del robot, y permiten a éste realizar un cálculo aproximado de la distancia que los separa a determinados obstáculos del entorno. Estos sensores pueden detectar objetos en un rango que va desde 25cm a 2.55m de proximidad, por debajo de este rango no existe una medición precisa de la distancia al objeto, simplemente puede detectarse la presencia de un cuerpo extraño, pero no detallar las coordenadas de éste.

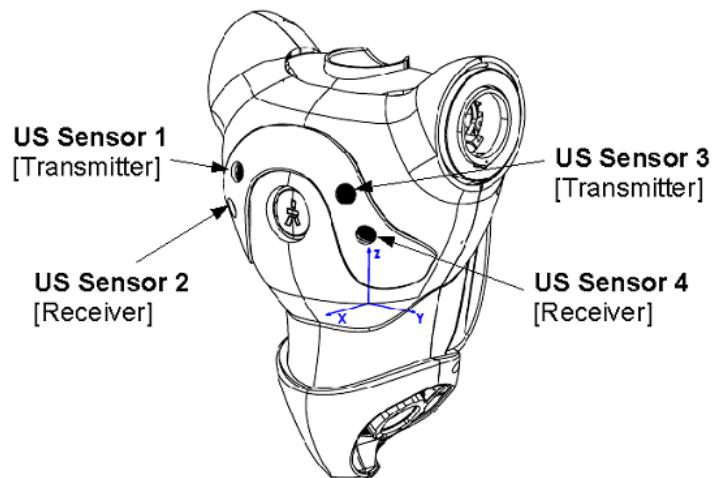


Figura 2.10: Posición de los sensores de ultrasonidos, situados en el torso del Nao.

## Cámaras

Son dos, ambas situadas en la zona de la cabeza de Nao, con ángulos de visión diferentes y no solapados, pero con especificaciones técnicas idénticas. Ambos dispositivos son cámaras OV7670, trabajando sobre el estándar gráfico VGA a una resolución de 640x480px, con una tasa de refresco de 30 imágenes por segundo (fps) y enfoque fijo. Un aspecto importante es que las cámaras emiten imágenes codificadas en el espacio de color YUV422, por lo que habrá que implementar conversiones de formato a nivel software si queremos trabajar en otros espacios de color, por ejemplo para las detecciones de objetos usando librerías OpenCV. Otro aspecto importante es la necesidad de jugar con los ángulos y el movimiento de la cabeza del robot, ya que como puede verse en la figura 2.11, existe una zona muerta entre los rangos de visión de las dos cámaras.

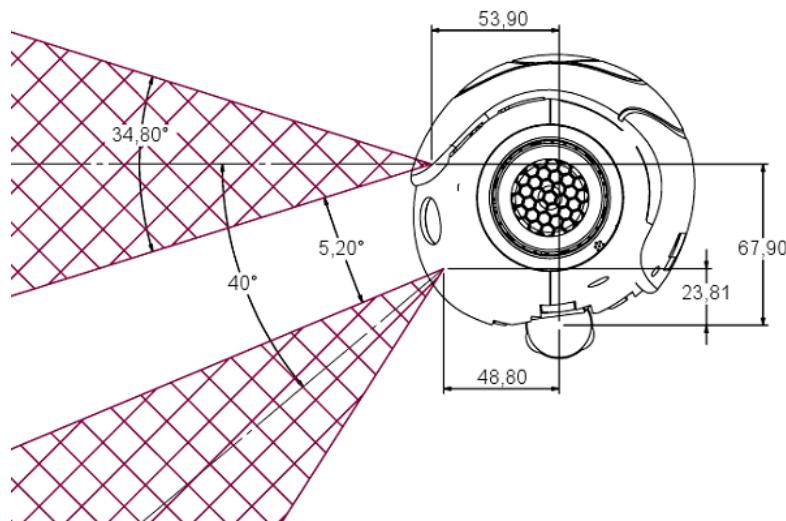


Figura 2.11: Posición y ángulos efectivos de las cámaras del Nao

## Hardware de propósito general

El procesador que incluido en Nao es un AMD Geode, una CPU con un solo núcleo, frecuencia de reloj de 550 MHz, arquitectura x86 con set de instrucciones de 32bit. Nao dispone además de 256 MB de memoria RAM, al que se unen 2GB de memoria tipo

flash dedicada a usuario.

### 2.2.3. Software del Nao

Aldebaran Robotics proporciona varios paquetes de software al usuario, plenamente funcionales por separado, pero aptos para el trabajo conjunto.[15] En el nivel más bajo, se encuentra *NaoQi*, que constituye el núcleo de software que maneja al robot, y que proporciona una librería sobre la que el resto de paquetes trabaja. Proporcionando un entorno de programación del robot a un nivel más alto, se encuentra *Choregraphe*.

#### Choregraphe

Choregraphe es una aplicación multiplataforma pensada como un entorno gráfico agradable al usuario, que nos permite diseñar movimientos y animaciones de forma interactiva. Sin embargo este entorno no permite el trabajo a bajo nivel con las librerías de *NaoQi*, ni facilita ninguna forma de integración con *ROS (Robot Operating System*, que se documentará en capítulos posteriores) por ello se ha prescindido de ella durante el desarrollo del proyecto.

A pesar de ello, cabe destacar que sí hemos usado Choregraphe para tener una visión de la cámara del Nao de forma inmediata en las simulaciones con el robot real, como se verá en el capítulo 9, *Pruebas*

#### NaoQi

*NaoQi* es un *framework* que proporciona una serie de librerías a través de las cuales podemos controlar el robot. Se trata de un sistema distribuido que permite una comunicación homogénea entre los diferentes sistemas hardware y módulos lógicos (movimiento, visión, audio, detección de obstáculos, etc.) de los que dispone el robot. Es un sistema multiplataforma (GNU/Linux, MacOS y Windows) y que puede ser llamado desde rutinas escritas en diferentes lenguajes de programación (mayormente C++ y Python).

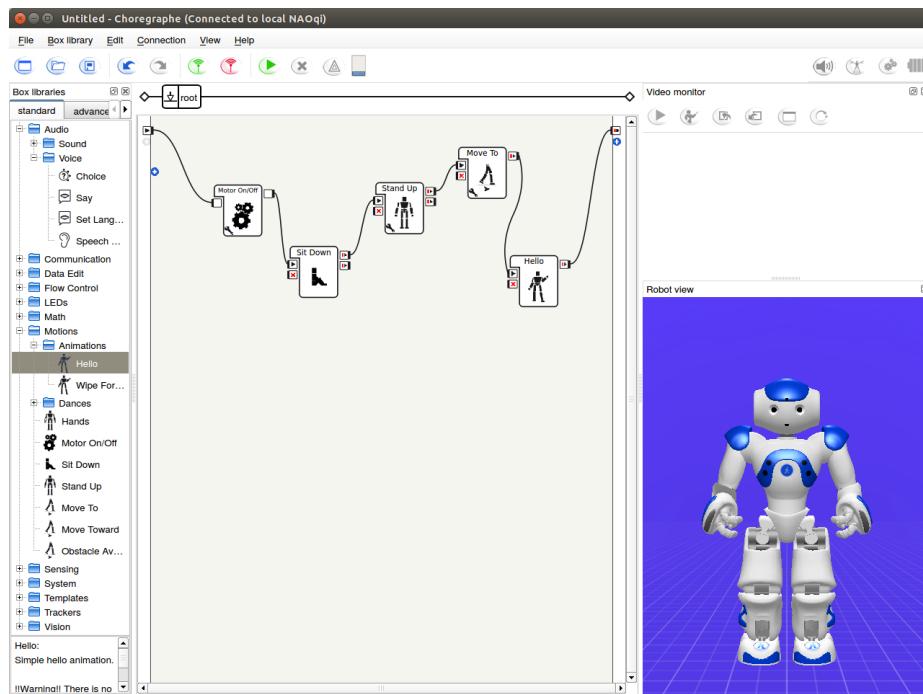


Figura 2.12: Choregraphe en funcionamiento.

La documentación de NaoQi hace una división lógica de las librerías que conforman la API. *Core* contendría aquellos módulos que gestionan a bajo nivel el hardware, como por ejemplo *ALMemory*, que permite administrar directamente la RAM del robot. *Motion* contiene módulos que facilitan la adecuación de Nao a un determinado movimiento requerido por el programador, como por ejemplo el movimiento sencillo de un brazo o una pierna del robot, mediante la definición de una trayectoria en el espacio. *Audio*, *Vision* y *Sensors* engloban los módulos que gobiernan el procesamiento de sonidos, imágenes y ultrasonidos, respectivamente.

Cada uno de los módulos de Nao, a su vez, está adherido a un *broker*. Un broker no es más que una construcción objectual que actúa como proveedor de servicios de direccionamiento, de tal forma que sea posible llamar a un método de un módulo de manera local o remota (en otro proceso e incluso en otra máquina) sin tener que llevar controles exhaustivos de acceso. El proceso de comunicación entre las diferentes entidades lógicas que conforman Naoqi, puede apreciarse en la figura 2.12.

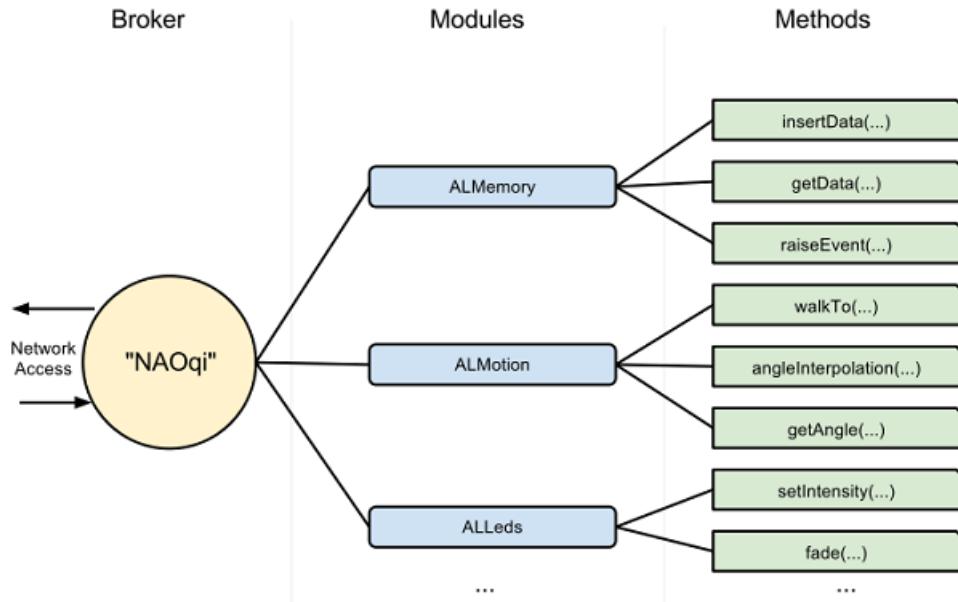


Figura 2.13: Diagrama jerárquico de NaoQi

Un aspecto importante a la hora de trabajar con NaoQi, es su diferenciación entre lo que se denomina *blocking calls*, y *non-blocking calls*, es decir, que los métodos que podemos llamar desde cualquier punto del código pueden suspender la ejecución del resto de procesos de Nao en ese momento (*blocking calls*), o bien pueden crear un proceso paralelo (*non-blocking calls*). Es decisión de diseño si una llamada a un determinado método debe aplicar un bloqueo o no.

En la figura 2.13 podemos ver un ejemplo práctico del uso de un método con bloqueo, en el que el flujo del proceso origen se detiene temporalmente, y no se reanuda hasta que el método iniciado mediante una llamada bloqueante termine su ejecución. El proceso contrario puede apreciarse en la figura 2.14, donde el primero de los procesos no detiene su ejecución en ningún momento.

Cada uno de los módulos de NaoQi puede ser instanciado. Cuando se crea una instancia de un objeto tipo módulo, éste, en terminología NaoQi, se denomina *proxy*. Cuando una variable tipo proxy se crea, es necesario especificar la IP que está asignada

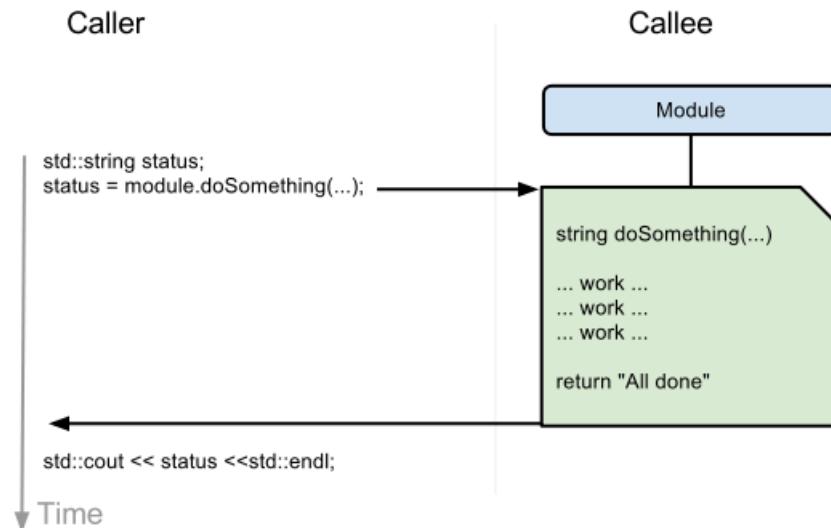


Figura 2.14: Llamada a método con bloqueo

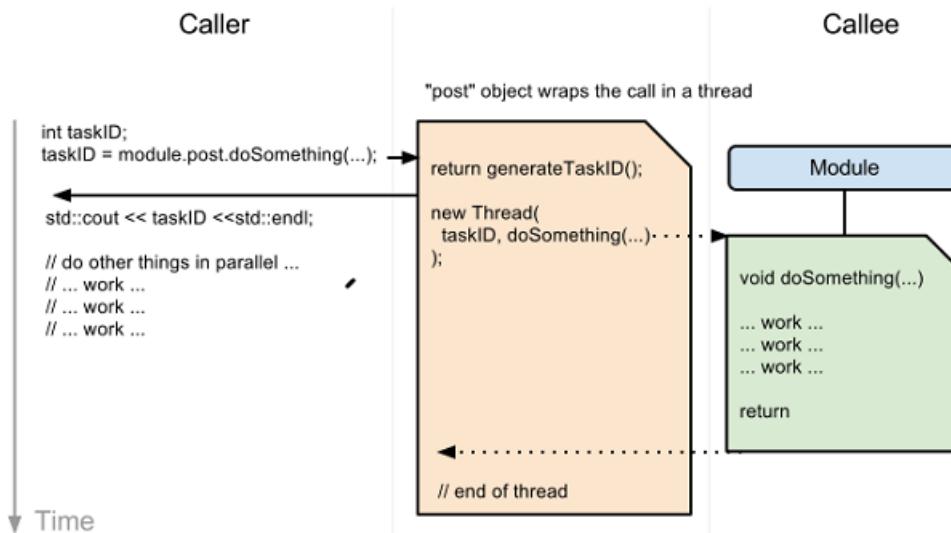


Figura 2.15: Llamada a método sin bloqueo o en paralelo

dentro de la red NaoQi a su broker. Tras ello, todo el trabajo de direccionamiento es realizado por el mismo broker.

Así, pueden distinguirse dos tipos de proxies en NaoQi:

- *Proxies locales*, que están lanzados sobre un mismo proceso y se comunican a través del broker padre de ambos.
- *Proxies remotos*, cuyo direccionamiento es, como se ha comentado, invisible al programador, pero en contrapartida trabajar con proxies en remoto hace que no podamos tener acceso rápido al espacio de variables (acceso directo a la memoria de Nao desde otro proceso externo por ejemplo).

Naoqi, en conclusión, puede ser entendido como un conjunto de librerías que permite la programación a bajo nivel del hardware del robot. Una vez que Naoqi se ejecuta, tendremos que generar una instancia de un módulo, es decir, un proxy. Mediante este proxy podremos acceder a los métodos del módulo. Un ejemplo sencillo del funcionamiento de Naoqi[16], que implementa en lenguaje de programación Python el que el robot se coloque en una posición erguida y camine 20 cm hacia adelante, se expone a continuación:

```
# Importamos librerías
from naoqi import ALProxy

# Generamos un proxy al módulo de movimiento.
motionProxy = ALProxy('ALMotion', 127.0.0.1, 9559)

# Generamos un proxy al módulo de posición corporal.
postureProxy = ALProxy('ALPosture', 127.0.0.1, 9559)

# Usamos las funciones de adoptar postura y de caminar.
postureProxy.goToPosture('StandInit', 0.5)
motionProxy.walkTo(0.20, 0, 0)
```

## 2.3. Robot Operating System (ROS)

*ROS*[6] es un *framework* que provee librerías y herramientas para el desarrollo de software orientado a robots. ROS provee abstracción de hardware, controladores para diversos dispositivos, herramientas de visualización de datos, comunicación intraprocisos, administración de paquetes, y, en gran medida, casi todas las funcionalidades que se podrían esperar de un sistema operativo al uso. Incorpora además herramientas de compilación de aplicaciones en diferentes lenguajes. ROS se distribuye bajo licencia BSD Open-Source. En esta sección se introduce el sistema de ficheros con el que ROS trabaja, así como su sistema de comunicación entre los diferentes elementos lógicos de ROS[17].

### 2.3.1. Sistema de ficheros de ROS

La unidad básica sobre la que ROS trabaja es el *paquete*. La gestión es muy sencilla: existen paquetes y ficheros de configuración de esos paquetes. Cada paquete tiene asociado un fichero (llamado *package.xml*) que detalla una descripción para el paquete, definiendo dependencias con otros paquetes del sistema de archivos y documentando números de versión, desarrollador del paquete, tipo de licencia aplicable al paquete, etc. Los paquetes tienen una estructura interna fija marcada por los estándares de ROS, y engloban una serie de elementos que lo componen, entre los que destacan:

- **Ficheros *package.xml*:** como se ha dicho, dependencias y metainformación del paquete.
- **Ficheros *launch*:** ficheros que especifican las condiciones de lanzamiento de un determinado paquete. Son útiles para, por ejemplo, detallar una secuencia de ejecutables que se deben lanzar (cada paquete puede estar formado por varios módulos) o para modificar las variables de entorno del sistema operativo antes de que un determinado ejecutable se lance.
- **Ficheros *msg*:** definen mensajes (ver sección 2.3.2).
- **Ficheros *srv*:** definen servicios (ver sección 2.3.2).

- **Ficheros de código fuente:** dentro de la organización jerárquica de un paquete, los fichero que contienen el código de una determinada rutina, pueden estar almacenados en la carpeta `/src` o bien en la carpeta `/scripts`, dependiendo de si el código es C++ (compilado) o Python (semi-interpretado), respectivamente.

Cabe destacar también que ROS posee un sistema de compilación (en inglés, *build system*) propio. Durante varias versiones de ROS, *rosbuild* fue el build system, pero en las últimas versiones los desarrolladores han diseñado *catkin*. La principal razón para implementar un sistema de gestión de código propio es dotar al sistema de consistencia y simplicidad. Los paquetes de ROS pueden depender a su vez del interprete Python, CMake, GTest o compilador GNU C++ (g++), entre otros. Catkin unifica todos estos entornos de desarrollo. Además catkin gestiona los mensajes y servicios de cada proceso, hace comprobaciones sobre los ficheros de lanzamiento, y supervisa la resolución de dependencias.

### 2.3.2. Sistema de comunicación y computación de ROS

Para la comprensión del sistema de comunicaciones de ROS, es indispensable introducir el concepto de *nodo*. Un nodo ROS es, en definitiva, un proceso, definido por un código fuente, y lanzado a través de un determinado paquete. El sistema de computación y comunicación de ROS funciona como una red uno-a-uno (en inglés, *peer-to-peer*) de procesos débilmente acoplados, que intercambian mensajes y se proporcionan y reclaman servicios entre ellos. Como consecuencia de esto, se puede hablar de dos tipos de comunicación en ROS: síncrona y asíncrona. Para el intercambio de solicitudes de servicios, ROS usa un sistema de comunicación síncrono de tipo RPC (en inglés, *remote procedure call*), mientras que para el intercambio de mensajes el sistema es asíncrono.

Más en detalle, podríamos hablar de diferentes entidades lógicas que intervienen en el sistema de comunicación:

- **Nodo:** como se ha dicho, es la unidad lógica básica del sistema ROS. Se trata de un proceso o rutina, escrito en C++ o en Python. Los nodos escritos en C++

dependen de la librería estándar incluida en la distribución ROS *roscpp*, mientras que los nodos cuyo código está escrito en Python lo hacen de *rospy*. Cuando un nodo ROS ha sido lanzado, puede enviar o recibir mensajes a través de un *topic*.

- **Topic:** un *topic* podría verse de forma abstracta como un banco de comunicaciones asíncrono. Un nodo publica mensajes sobre uno de los topics activos en ese momento, de tal forma que dichos mensajes pueden ser distribuidos a través de la red de nodos de ROS. El nodo que publica los mensajes suele denominarse, por influjo del inglés, *publisher*, mientras que el nodo que lee los mensajes que están siendo publicados en el topic, recibe el nombre de *subscriber*. Cabe destacar que existe la posibilidad de que un mismo topic reciba mensajes de varios publishers, así como varios subscribers pueden leer mensajes publicados sobre el mismo topic, definiendo una relación muchos-a-muchos entre los nodos.
- **roscore:** es el conjunto de nodos y programas que constituye la base del sistema ROS, y que controla la interacción del resto de nodos y servicios lanzados sobre la red. Roscore incluye *ROS Master*, que ejerce de proveedor de servicios de registro de nombres y controla las comunicaciones. Incluye también *rosout*, un mecanismo de *logging* propio que asimismo incorpora herramientas de visualización de datos.
- **Paramenter Server:** diccionario dinámico y compartido de variables, al que los nodos de ROS pueden acceder en tiempo de ejecución. Por lo general usado para almacenar parámetros de configuración generales. Está implementado para la comunicación síncrona entre diferentes partes de una red ROS, usando un protocolo de transporte de tipo RPC-XML.<sup>1</sup>
- **Mensaje:** un mensaje en ROS es un conjunto de información encapsulada cuyo formato está definido por un fichero *msg*. Un mensaje puede consistir en datos de tipo primitivo (integer, float, boolean, etc.) o bien formado tipos complejos, en forma de arrays y estructuras, e incluso anidando éstas. Una vez que el programador define un tipo de mensaje, éste debe ser generado (compilado) usando el *build-system* estándar de ROS que ya se ha introducido, *catkin*, y podrá ser usado

---

<sup>1</sup>véase Apéndice A.

en cualquier nodo ROS. Cabe destacar que ROS incorpora un tipo de mensaje estándar, *Header*, que encapsula ciertos metadatos como un registro temporal y un identificador numérico de mensaje.

- **Servicio:** En combinación con el paradigma nodal de comunicación asíncrona, tenemos en ROS un tipo de comunicación síncrona en dirección de doble sentido, los servicios. Los servicios son tránsitos de información en dos fases, del tipo petición/respuesta, proporcionados por uno de los nodos de ROS, que ejercería de destinatario de las peticiones, y que emitiría las respuestas. Los servicios están definidos dentro de ficheros *srv*, y al igual que los mensajes, son compilados por *catkin* previamente a la ejecución de los nodos que van a hacer uso de ellos.
- **Bags:** mecanismos de almacenamiento de mensajes. Además de las herramientas de *logging*, ROS incorpora este tipo de entidades, de las que los nodos pueden obtener mensajes emitidos con anterioridad en tiempo de ejecución.

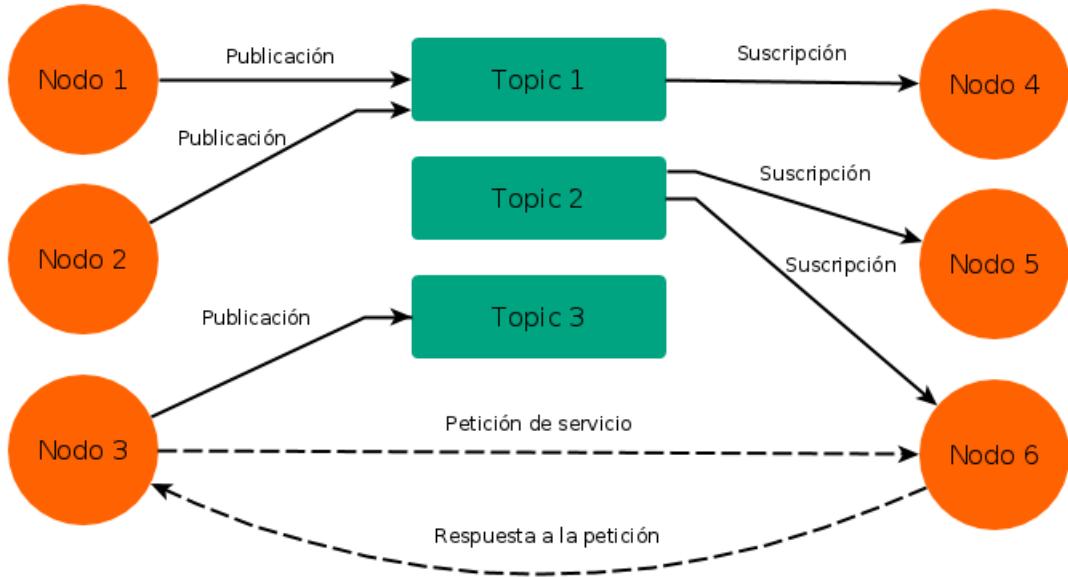


Figura 2.16: Sistema de comunicación ROS sencillo

Como puede verse en el diagrama de la figura 2.15, es posible que cierto topic no tenga suscriptores (véase Topic 3). En este caso los mensajes se almacenan en un buffer

(como corresponde a un paradigma de comunicación asíncrona) cuyo tamaño máximo puede ser especificado por el programador. También es posible que aunque haya varios nodos suscritos, ningún nodo haya comenzado a publicar en un topic (véase Topic 2). Cuando asignamos suscripciones a un topic, ROS lo implementa automáticamente aunque no se estén publicando mensajes sobre él.

### Otros aspectos del proceso de intercomunicación en ROS

Además de la comunicación entre nodos simples, cada uno de los nodos posee una línea de comunicación permanente con el nodo *master*. Éste maneja el registro del espacio de nombres de ROS, de tal forma que los nodos no podrían encontrarse sin los servicios proporcionados por el nodo master. Sin embargo, una vez que el direccionamiento está resuelto, los nodos se conectan de forma directa. En este sentido, podría pensarse en el nodo master como una suerte de servidor DNS, de hecho, el protocolo de comunicación ROS es *TCPROS*, que usa sockets TCP/IP para la transmisión de paquetes optimizados para ROS. El direccionamiento a través del master es una tarea crítica, ya que cualquier librería-cliente (rospy, roscpp, etc.) puede hacer cambios en el espacio de nombres (nodos, topics, servicios, etc.) en tiempo de ejecución, por lo que el mapa de nombres debe estar constantemente actualizado para evitar inconsistencias y errores de computación.

Cabe destacar también que el modelo de ROS implica un desacoplamiento total entre los procesos que están lanzados en el entorno. Es decir, los nodos publican y reciben mensajes, pero son completamente independientes; pueden ser terminados, iniciados o reiniciados en cualquier orden sin inducir a estados de error ni a generar inestabilidad en el sistema.

## 2.4. Estado del arte

En esta sección se introduce brevemente alguno de los estudios y trabajos a los que se ha tenido acceso, y están relacionados con el campo del presente documento. Además, de forma bastante genérica, se expondrá el fundamento de la Robocup, com-

petición basada en la investigación con robot que ha servido de motivación e inspiración.

#### **2.4.1. Diseño del comportamiento de un portero de fútbol robótico**

Existe un trabajo muy relacionado con el que se ha proyectado: **Diseño del comportamiento de un portero de fútbol robótico**[18], desarrollado por Rubén Vegas Cañadilla de la Universidad Rey Juan Carlos. Como puede extraerse del título, el proyecto gira en torno a la idea que establecer un comportamiento sobre el robot Nao, que permita a éste detectar una pelota, y, siendo consciente de la línea que marcaría el gol, desplazarse lo suficientemente rápido como para que la pelota en movimiento no la traspase. El proyecto de Rubén Vegas cubre varios objetivos muy interesantes como el posicionamiento del robot en función del área de gol, la alineación con respecto a la pelota, la detección de ésta en un entorno dinámico en tiempo real, o el trazado de desplazamientos que permitan al robot interponerse en la trayectoria de la pelota de forma automática.

#### **2.4.2. Construcción y pruebas en Gazebo y V-Rep de un modelo para el robot humanoide Nao**

Podríamos citar también este trabajo de Davide Zanin[19], tesis publicada por la Universidad de Padua, que adapta las texturas del modelo URDF (Unified Robot Description Format) proporcionado por el propio fabricante de Nao para la creación de un modelo válido para el simulador libre V-REP y también extrapolable a Gazebo. Además propone la integración en ROS de dicho modelo, y hace un estudio exhaustivo de los simuladores Gazebo y V-REP, ambos considerados para la inclusión en este proyecto de fin de carrera. Mide experimentalmente los resultados y la eficiencia de diversos movimientos del robot en los citados entornos de simulación, incorporando el modelo 3D adaptado. Este proyecto no tiene una parte práctica de diseño de movimientos y relaciones con el entorno, pero sirve como guía a la hora de trabajar con las texturas 3D que proporciona Aldebaran.

### **2.4.3. Aprendizaje por Refuerzo Seguro para enseñar a un robot humanoide a caminar más rápido**

Cabe destacar también este texto[20] de la Universidad Carlos III de Madrid, desarrollado por Daniel Acera Bolaños. Este trabajo se centra en estudiar los patrones de movimiento de Nao cuando camina, de tal forma que mediante un algoritmo de aprendizaje se pueda intentar mejorar la velocidad al caminar. La arquitectura que se propone está también incluida en ROS. Se hace un estudio exhaustivo del movimiento caminar que incorpora Nao por defecto, y se propone mediante ciertos algoritmos de aprendizaje por refuerzo seguro la modificación automática de los parámetros de los drivers proporcionados por ROS para el caminar del robot. Cabe destacar que el desplazamiento estudiado es solamente el que se corresponde con el frontal, de hecho el autor cita como objeto de estudio futuro una adecuación del algoritmo de aprendizaje para desplazamientos omnidireccionales.

### **2.4.4. Robocup**

Uno de los aspectos más atractivos cuando se trabaja con robots, es explorar la capacidad de éstos para moverse por el entorno respondiendo a unos parámetros determinados por las condiciones del entorno, de tal forma que emitan unas respuestas (movimientos) a estímulos de forma dinámica. La Robocup que se celebra en el año 2014 en Brasil, al igual que el mundial de fútbol[21]

Encuadrada dentro de la robótica móvil está la *RoboCup*, una competición internacional que tiene como finalidad promover la investigación y desarrollo de la inteligencia artificial. Esta competición se basa en enfrentar a dos equipos de robots completamente autónomos para que jueguen un partido de fútbol. Nao es el robot oficial de la Robocup desde 2008.

Algunos de los comportamientos específicos para la Robocup que han sido imple-

mentados usando el robot Nao, pueden verse online. Como ejemplo, podemos citar el equipo *Berlin United*[22], cuyo golpeo de pelota puede apreciarse en el portal Youtube.

<sup>2</sup>



Figura 2.17: Nao en el terreno de juego de la Robocup.

---

<sup>2</sup><https://www.youtube.com/watch?v=4dHeKAql99I>



## **Capítulo 3**

# **Valoración de hardware y software**

Durante la realización de este proyecto, y debido a la inexperiencia del autor en el campo del que trata, se ha tenido que dedicar un tiempo notablemente extenso a la familiarización con los elementos de software que lo soportan, y a su vez a la toma de decisiones asociada. A lo largo de varios meses, se han tenido que tomar varias decisiones críticas sobre los componentes a usar. En este capítulo se expone una justificación de cada una de las decisiones tomadas en este sentido.

### **3.1. Tecnologías utilizadas**

A continuación se expondrá una lista general de todo el hardware y software utilizado, y en las siguientes secciones se detallarán aquellos componentes que no sean de propósito general o bien hayan generado una toma de decisión por parte del autor, en cuyo caso se indicará también una justificación de la misma.

<b>Hardware</b>
PC portátil Sony Vaio VPCF22
Nao Robot

Tabla 3.1: Lista de hardware utilizado durante la realización del proyecto.

<b>Software</b>	<b>Tipo de licencia</b>
Ubuntu GNU/Linux 14.04 LTS	GNU Public License
ROS Indigo	GNU Public License
Webots EDU 7.4.3	Propietaria
NAOqi 1.14.3	Propietaria
Python 2.7.6	GPL-compatible
GCC 4.8 compiler	GNU Public License
Sublime Text 2.0.2	Propietaria
yEd Graph Editor 3.12.2	Propietaria
pdfTeX 3.1415926-2.5-1.40.14	GNU Public License
GeoGebra 4.0.34	GNU Public License

Tabla 3.2: Lista de software utilizado durante la realización del proyecto.

### 3.2. Sistema operativo local

Durante el proceso completo de realización del proyecto se ha empleado el sistema operativo Ubuntu 14.04 LTS. Se ha valorado la utilización de otros sistemas operativos libres basados en GNU/Linux, pero finalmente se ha optado por usar Ubuntu debido principalmente a su integración con otras partes del proyecto, como por ejemplo ROS, Gazebo o Webots. De hecho en el caso de ROS, sus propias librerías se basan en las de Ubuntu, por ello la integración es total, y la instalación es sencilla y natural, al igual que la instalación de nuevos paquetes y funcionalidades.

Además, debido a la gran dependencia que el desarrollo del proyecto tiene de un simulador gráfico, y motivado por el hardware del equipo en el que se desarrolla, Ubuntu proporciona controladores propietarios desarrollados por la compañía Nvidia, por lo que el rendimiento en este sistema operativo de los simuladores 3D valorados es óptimo.

### 3.3. Sistema operativo de gestión del robot (ROS)

Aunque como ya se ha visto, desde el punto de vista estricto, el núcleo de Nao podría decirse que el conformado por Naoqi, se ha optado por ROS para gestionar las comunicaciones entre las diferentes entidades lógicas que están presentes en el robot. Lo cierto que es la utilización del sistema operativo ROS como gestor de Nao está presente como premisa en el anteproyecto, pero de todas formas se lleva a cabo un proceso de validación de este software y de estudio de sus posibilidades. Para ello se tomaría como referencia tanto la documentación oficial de ROS, disponible en su *wiki* oficial, como el libro *Learning ROS for Robotics Programming*[5]

#### 3.3.1. Justificación de uso de ROS

ROS provee un sistema completamente modular y distribuido, esto quiere decir que a la hora de aplicarse a un proyecto de robótica concreto, el diseñador tiene la posibilidad de elegir la infraestructura que va a tener ROS en la arquitectura final. Esta es una de

las razones de que ROS sea una opción ideal: podemos prescindir de sus funcionalidades donde nuestro sistema sea más eficiente llamando directamente a los controles de bajo nivel del robot (Naoqi) y podemos echar mano de cualquiera de las múltiples herramientas que ROS proporciona cuando consideremos que esta es la solución más óptima.

### Valoración del software

El objetivo es la adquisición de los conocimientos teóricos y prácticos que permitan al autor gestionar un sistema robótico completo usando ROS. Como objetivo se fija también el dominio del transporte de mensajes a través de ROS, programando desde cero *publishers*, *subscribers* y *servicios* de ROS, viendo su implementación tanto usando el lenguaje de programación C++ como Python. Una vez que se han adquirido estas habilidades, el autor está en disposición de elegir qué funcionalidades de ROS se aplicarán al proyecto, así como comprobar la viabilidad de utilización de este software para el caso concreto del proyecto.

### Pruebas sobre ROS

Se diseñan diferentes nodos de ROS. Más concretamente se diseña a partir de los ejemplos encontrados en la bibliografía, un *publisher* simple, que emita sobre un *topic* de prueba un mensaje con una cadencia de 10Hz, así como un receptor de dichos mensajes, un *subscriber* que devuelva un eco que podamos visualizar desde la terminal desde la que se lanza el nodo. La implementación de estos nodos se hizo tanto en Python como en C++, aunque posteriormente en el proyecto se eligió Python como lenguaje en casi todas las implementaciones de nodos en las que finalmente consistió el sistema. En el siguiente fragmento de código puede verse la implementación del *publisher* en Python:

```
import roslib;
import rospy
from std_msgs.msg import String
def emisor_prueba():
```

```

pub = rospy.Publisher('topic_prueba', String)
rospy.init_node('emisor_prueba', anonymous=True)
r = rospy.Rate(10) # 10hz
while not rospy.is_shutdown():

    str = "hello world %s"%rospy.get_time()
    rospy.loginfo(str)
    pub.publish(str)
    r.sleep()

if __name__ == '__main__':
    try:
        emisor_prueba()
    except rospy.ROSInterruptException: pass

```

Y el correspondiente *subscriber*, escrito en esta ocasión en C++:

```

#include <ros/ros.h>
#include <std_msgs/String.h>
void receptor_callback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("Eco: [%s]", msg->data.c_str());
}
int main(int argc, char **argv)
{
    ros::init(argc, argv, "receptor_prueba");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("topic_prueba", 1000, receptor_callback);
    ros::spin();
    return 0;
}

```

### 3.3.2. Justificación del uso de Python para implementar nodos ROS

Tanto ROS como Naoqi, que se introducirá en la sección aceptan como lenguajes de programación nativos C++ y Python. En la realización de este proyecto, se ha empleado Python 2.7.6.

La decisión ha venido motivada por la propia naturaleza del lenguaje: Python es un lenguaje semi-interpretado, que permite la ejecución de órdenes de forma interactiva a través de un intérprete. Se antoja como muy deseable, en tanto estamos realizando un proyecto cuyas pruebas requieren de cierto tipo de interacción con el entorno, la capacidad para crear de forma dinámica scripts y poder apreciar también dinámicamente el resultado de la implementación. Ello, unido a la plena integración de Python en Naoqi y en ROS<sup>1</sup>, hicieron que se tomase esa decisión desde un primer momento.

### 3.3.3. Conclusiones

Como resultado de la implementación de estos experimentos, se observa un pequeño sistema ROS en plena ejecución: se puede comprobar la utilidad de éste a la hora de propagar información a través de un sistema de comunicaciones distribuido. Para ello, se hace uso en la exploración de las múltiples herramientas de visualización de datos que incorpora ROS, como por ejemplo *rxgraph*, que nos permite comprobar en tiempo real nuestro diagrama de comunicaciones.

Observamos por tanto que ROS es una excelente forma de coordinar los diferentes módulos en los que se dividirán cada una de las funciones que realice nuestro robot. ROS facilita el intercambio de mensajes y la visualización de resultados a tiempo real, así como provee de herramientas útiles de logging y gestión de código fuente. Se confirma, por tanto, mediante pruebas experimentales, la viabilidad y deseabilidad de su uso para este proyecto.

## 3.4. Sistemas de representación de objetos 3D

Durante la preparación del entorno de desarrollo, se han tenido en cuenta diferentes simuladores y mecanismos de representación del robot en un espacio tridimensional.

---

<sup>1</sup>ROS incluso cuenta con un cliente nativo de Python, cuya documentación puede consultarse en <http://wiki.ros.org/rospy>

nal, que permita la rápida evaluación de los diferentes pasos de aproximación a los comportamientos-objetivo. En esta sección se detallarán los modelos de representación visual considerados: URDF y SDF.

### 3.4.1. Unified Robot Description Format (URDF)

URDF es una especificación, derivada de XML, para la descripción de estructuras robóticas. Cubre tres aspectos fundamentales: cinética de la estructura, representación visual del conjunto, y definición del espacio de colisiones del robot. ROS se encarga del procesamiento de las definiciones escritas usando la sintaxis de URDF. Para ello hace uso de diferentes módulos que, en última instancia, buscan la comunicación directa con el sistema operativo bajo el que corre ROS, que será Ubuntu en nuestro caso. Un diagrama de esta interacción y de sus diferentes partes se muestra en la figura 3.1

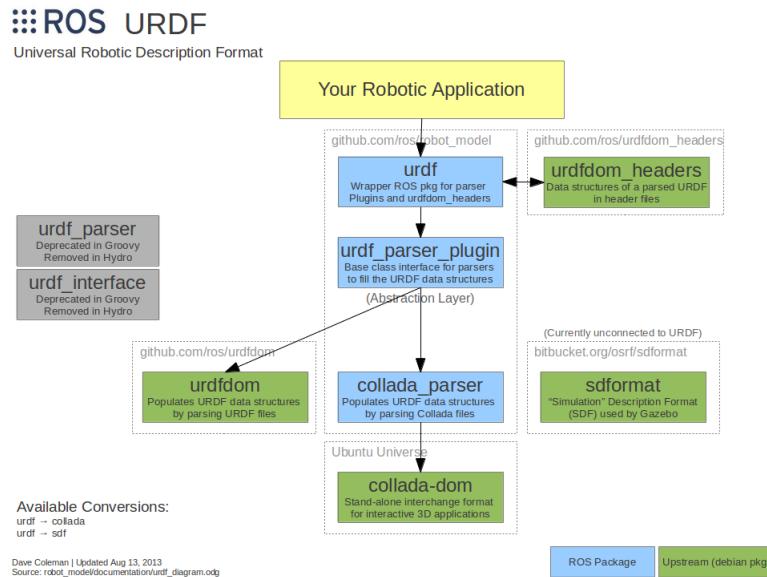


Figura 3.1: Conjunto de paquetes ROS que proporcionan la interpretación de URDF

Existen varios elementos de los que puede componerse una especificación URDF. Algunos de los más importantes son los *links*, que representan partes inmóviles del robot, *joints*, que hacen de unión entre links, o *transmissions*, que definen la unión entre un joint y un determinado motor del robot. Cada uno de los elementos de URDF tiene unas propiedades determinadas que puede ser definidas. La especificación completa puede consultarse en la documentación de ROS, que se referencia en la bibliografía.

A modo de ejemplo, se detalla un diseño, con dos *links* y un *joint* que los une:

```
<robot name='robot\_ejemplo'>
  <link name='link1' />
  <link name='link2' />
  <joint name='joint1' type='continuous'>
    <parent link='link1' />
    <child link='link2' />
    <origin xyz='5 3 0' rpy='0 0 0' />
    <axis xyz=' -0.9 0.15 0' />
  </joint>
</robot>
```

### 3.4.2. Simulation Description Format (SDF)

SDF es la especificación en un principio creada para el simulador Gazebo (ver sección 3.4.1) que conserva la mayor parte de la sintaxis de URDF, pero que extiende ésta en ciertos aspectos. Está, al igual que URDF por tanto, basada en etiquetas XML. Uno de los principales aspectos de SDF es que, al contrario de lo que ocurría con URDF, puede especificar varios robots, un entorno, y las condiciones de éste. La especificación completa de los distintos elementos que forman SDF puede consultarse en la documentación de Gazebo, que se referencia en la bibliografía.

Un diseño mínimo escrito en SDF, y obviando las definiciones iniciales por simplicidad, podría ser:

```
<?xml version='1.0'?>
<sdf version='1.4'>
<model name='modelo\_minimo'>
```

```
<pose>0 0 0.5 0 0 0</pose>
<static>true</static>
<link name='link'>
  <collision name='collision'>
    <geometry>
      <box>
        <size>1 1 1</size>
      </box>
    </geometry>
  </collision>
  <visual name='visual'>
    <geometry>
      <box>
        <size>1 1 1</size>
      </box>
    </geometry>
  </visual>
</link>
</model>
</sdf>
```

### 3.5. Simuladores

Encontrar un sistema de simulación en tiempo real vía software es una tarea crítica en el diseño e implementación de comportamientos para un robot complejo. La implementación de movimientos siguiendo el paradigma de ensayo-error, o bien desde un punto de vista experimental, implica que numerosas operaciones estén en fase de ensayo, y, por tanto, son susceptibles de ocasionar caídas y diversos percances al robot. Aunque Nao cuenta con un sistema de equilibrio monitorizado por software, al trabajar directamente con órdenes de bajo nivel lanzadas sobre el API de Naoqi, se obliga en muchos casos al robot a comprometer su equilibrio inercial. Una alternativa a esta generación de desequilibrios podría ser Cartesian Control, que se expone en la sección 3.5.2.

Además, en nuestro caso, el sistema de representación y simulación debe considerarse crítico también por otra cuestión: hasta su implementación en el robot real y durante la simulación, el robot adquiere imágenes del entorno 3D generado por el propio simulador. Esto redunda en la importancia de poder contar con un entorno creíble, lo más cercano a la realidad posible, de tal forma que se suavice y se haga lo menos traumático para el sistema posible el paso posterior a trabajar en un entorno real, con cámaras reales tomando imágenes reales.

Los simuladores que se han considerado durante la elaboración del proyecto son Gazebo, V-REP y Webots. En las secciones siguientes se expone cada uno de ellos, se realiza una valoración de los mismos, y se concreta por qué se ha elegido Webots como el indicado para el proyecto.

Cabe destacar que Aldebaran ha venido proporcionando un simulador antes de incorporar Webots como predeterminado, pero debido a que sólo existe versión para Microsoft Windows, se que desestimado su uso desde un primer momento.

### 3.5.1. Gazebo

Gazebo[3] es un simulador de código abierto, capaz de representar en tiempo real, en un entorno tridimensional, varios robots, sensores y objetos. Además, implementa las relaciones físicas de estos objetos, detallando las tensiones entre los materiales que componen estos objetos, así como modelando el comportamiento dinámico de éstos en relación a las fuerzas físicas teóricas que influirían en ellos en el mundo real.

Gazebo utiliza el motor de renderizado 3D OGRE para proporcionar al usuario luces, sombras, y texturas de gran realismo gráfico. Incorpora además un API propio, que se referencia en la bibliografía del presente trabajo, y un conjunto de herramientas a través de la línea de comandos.

Como se introdujo en la sección 3.3.2, Gazebo utiliza de forma nativa el lenguaje de descripción de autómatas SDF, aunque también es capaz de importar modelos escritos siguiendo la sintaxis URDF. De hecho, incorpora el módulo *gzsdf*, capaz de efectuar traducciones desde un modelo URDF a uno SDF. Las librerías de la aplicación además contienen una colección de especificaciones de robots, entre los que no se encuentra, al menos en el momento de realización de este trabajo, la descripción del robot Nao.

Cabe destacar la posibilidad de escribir pequeñas porciones de código, que Gazebo reconoce como *plugins*, que brindan al programador de comportamientos la posibilidad de alterar la simulación de forma procedimental. Los plugins son compilados como código C++ usando la herramienta cmake, y posteriormente Gazebo los incorpora a la simulación. Pueden controlar modificaciones del entorno, creación de nuevos objetos o cambios en los comportamientos de éstos. Es un concepto que podría tener cierto paralelismo con los *controladores* de Webots, que se verán en la sección 3.4.3.

**Gazebo fue la alternativa sugerida en el anteproyecto** a la hora de visualizar de forma dinámica el robot en un entorno tridimensional generado por software, sin embargo a medida que el proyecto iba tomando forma se vió la imposibilidad de tener un sistema estable de representación en un tiempo reducido. Se detallan a continuación los aspectos más importantes del trabajo realizado con Gazebo.

Se trató desde un primer momento de crear un entorno sencillo pero creíble, que nos pudiese dar una idea del comportamiento del robot en el mundo real, así como que pudiese generar un contexto lo más similar posible a éste. Esto implica que tanto la representación del robot como la del mundo que lo rodea, debería ser completamente extrapolable a la realidad.

Existe un periodo de familiarización con el entorno visual que propone Gazebo, de familiarización con los controles básicos del programa que nos permiten manejar una

simulación. Tenemos además, que Gazebo incorpora herramientas de importación de modelos robóticos definidos bajo la sintaxis SDF, por lo que tendremos que proveernos de una representación de Nao correcta para trabajar. Como no existe ninguna especificación SDF proporcionada por el fabricante, tendremos que escribir una a partir del URDF que sí se proporciona a un nivel básico articular por mediación de Aldebaran Robotics, y, por su gran extensión, traducirla a SDF usando la herramienta ad hoc de Gazebo (*gzsdf*).

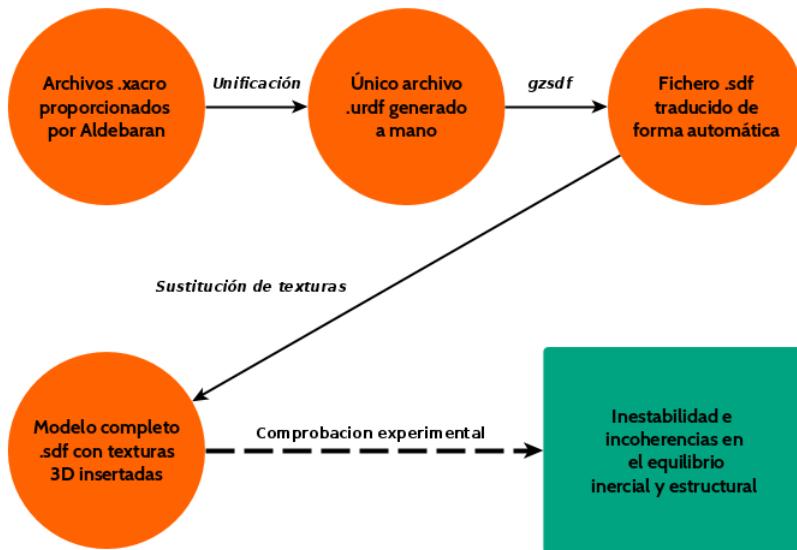


Figura 3.2: Diagrama del proceso de adecuación del modelo de Nao al simulador Gazebo

Cabe destacar que Aldebaran sí proporciona, y se incluye en el paquete integrado en ROS *nao\_description*<sup>2</sup>, una serie de texturas 3D básicas del Nao, de tal forma que se pueden adherir al modelo básico que ya se ha citado. De hecho, gran parte de trabajo en este aspecto se realizó intentando traducir el modelo original proporcionado por Aldebaran y descrito usando el *macro-lenguaje* propio de ROS *Xacro*<sup>3</sup> a un sólo código URDF, almacenado en un sólo fichero, para posteriormente hacer la traducción a SDF mediante la herramienta ya citada *gzsdf* y ahí adjuntar las texturas proporcionadas.

<sup>2</sup>[http://wiki.ros.org/nao\\_description](http://wiki.ros.org/nao_description)

<sup>3</sup><http://wiki.ros.org/xacro>

nadas por Aldebaran. El proceso completo se puede ver de forma gráfica en la figura 3.2.

En el siguiente código pueden verse diversos fragmentos del URDF oficial que proporciona Aldebaran Robotics y que se incluye en ROS, más concretamente la definición del *link* que conforma el torso, que consiste en un simple cilindro:

```
<link name='base_link' />
<link name='torso'>
    <inertial>
        <origin rpy='0 0 0' xyz='-0.00413 0.00009 0.04342' />
        <mass value='1.04956' />
        <inertia ixx='0.00487953284' ixy='-0.00001428591' ixz='-0.00019545651'>
            iyy='0.00470360698' iyz='0.00002224589' izz='0.0015671352' />
        </inertia>
    </inertial>
    <visual>
        <origin rpy='0 0 0' xyz='0 0 0' />
        <geometry>
            <mesh filename='package://nao_description/urdf/meshes/V32/Torso.mesh' scale='0.001 0.001 0.001' />
        </geometry>
        <material name='LightGrey' />
    </visual>
</link>
```

A pesar de que la utilización de Gazebo fue descartada por los motivos que se han expuesto, el fabricante del Nao, Aldebaran Robotics, sigue trabajando en el diseño de un modelo estable para el robot bajo la plataforma Gazebo.<sup>4</sup>

### 3.5.2. V-REP

V-REP es un simulador multiplataforma que permite la creación de controladores propios a través de una API nativa, principalmente desde código C++, aunque también existen wrappers para Python, Java o Matlab. Permite la composición de escenarios

---

<sup>4</sup><https://groups.google.com/forum/#!forum/ros-sig-aldebaran>

complejos e incorpora una extensa biblioteca de modelos de robots comerciales. Entre los modelos tridimensionales que se adjuntan en el propio simulador está Nao. Además, V-REP permite integración con ROS, posibilidad de gestionar el simulador mediante los paradigmas de la arquitectura de ROS (nodos, servicios, mensajes...).<sup>5</sup>

A pesar de que, como se ha comentado, V-REP cuenta con un modelo tridimensional propio para Nao, no está implementada, en el momento en el que se realiza este trabajo, una integración de Naoqi en el sistema. Es decir, el simulador no es capaz de conectarse al núcleo del robot, y no es posible trabajar con la API de Naoqi, ni generar código ROS que haga puente entre la API y la arquitectura. Debido a esto, se desestima el uso de este simulador. Para futuros proyectos, se hace notar que si se integra Naoqi en V-REP, el simulador a priori pasaría a ser una opción muy interesante para trabajar con Nao.

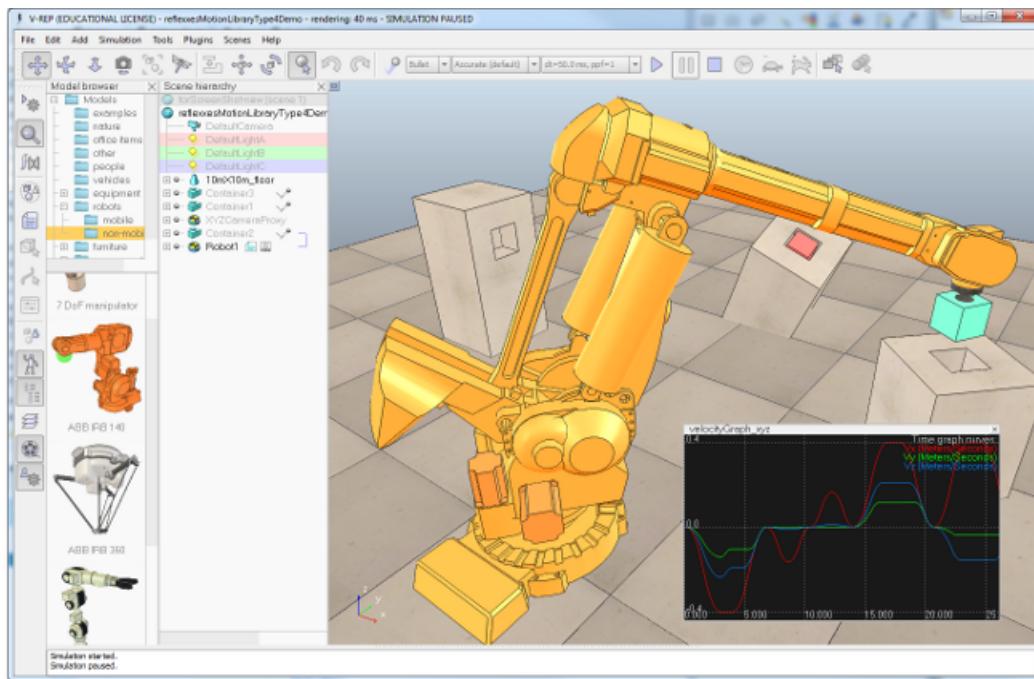


Figura 3.3: Interfaz gráfica y simulación en V-REP

<sup>5</sup><http://www.coppeliarobotics.com/features.html>

### 3.5.3. Webots

Webots[4] es un software propietario, desarrollado por la empresa suiza Cyberbotics. Webots es un entorno de diseño, modelado y simulación de robots. Permite la simulación de varios robots en un mismo entorno, manejando las propiedades físicas y los comportamientos programáticos de cada uno de ellos con total independencia.

Cada uno de los robots presentes en una determinada simulación, está determinado por su *controlador*. Un controlador una colección de código que maneja el robot, establece su comportamiento ante determinados estímulos, y gestiona su sistema de adquisición de datos del exterior. Los robots implementados en Webots pueden tener controladores propietarios, proporcionados por sus fabricantes, o pueden gestionarse mediante controladores escritos por el usuario. Para ello, éste puede hacer uso de varias API integradas en Webots: Java, Python, C, C++ e incluso Matlab.

En el caso de Nao, tenemos el controlador estándar, Naoqi, visto con detenimiento en la sección 2.2.3. Por ello, la integración es sencilla e instantánea: Webots se encarga de generar el modelo tridimensional integrado del robot, y lanza Naoqi para su gestión. De hecho, Webots es el software de simulación que Aldebaran Robotics incluye en las nuevas versiones de Nao bajo el nombre de *Webots for Nao*.

### 3.5.4. Conclusiones

Usando Gazebo con los modelos 3D URDF/SDF, la integración del modelo visual con las funcionalidades del controlador y del sistema operativo ROS no llegó nunca a niveles aceptables, por ello se desestimó la utilización de Gazebo como herramienta de trabajo. La utilización de Gazebo implica la necesidad de desarrollar un modelo 3D del robot, ya que no existe ningún modelo proporcionado por el fabricante, y tampoco ninguno disponible bajo una licencia de software libre, que permita la integración en el entorno Gazebo, en el momento de realización de este proyecto. Tras comprobar que dicho desarrollo a integración se sale de los objetivos globales de este trabajo, se desestima el trabajar con Gazebo.

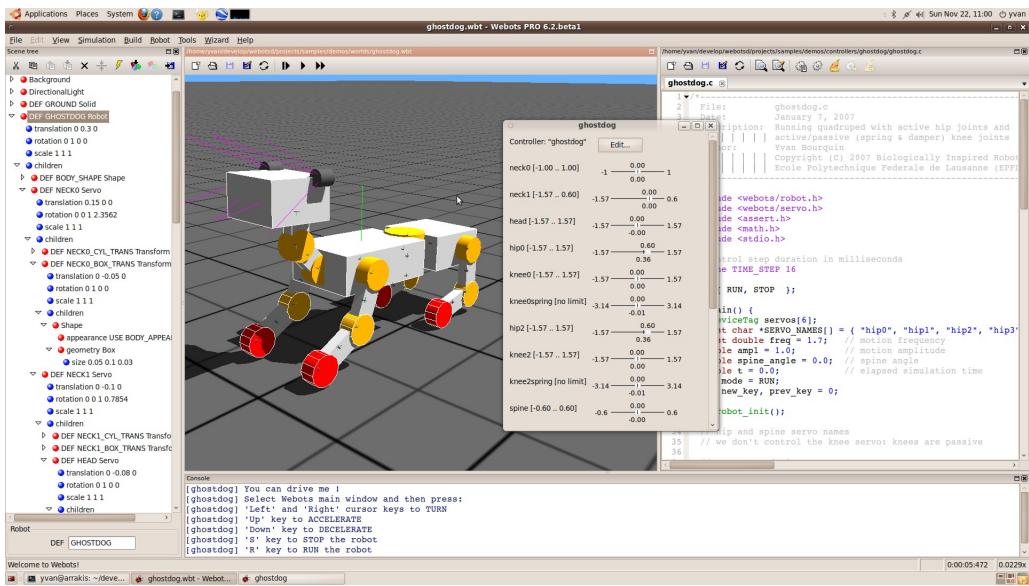


Figura 3.4: Interfaz gráfica y simulación mediante Webots

Asimismo y por las razones comentadas de incompatibilidad a la hora de controlar el robot simulador a través de la API de Naoqi, se desestima también la posibilidad de usar V-REP como simulador de trabajo.

Sin embargo, se comprueba experimentalmente que incorporado en el sistema Webots, tenemos:

- Una representación tridimensional del robot Nao precisa, estable y creíble.
  - Un entorno tridimensional y fácilmente modificable.
  - Una integración total con Naoqi, gestionada automáticamente por el propio simulador.
  - La capacidad de integrar controladores ROS independientemente al entorno de simulación.

Atendiendo a todas estas razones, y a pesar de tratarse de un software propietario bajo licencia, **se selecciona Webots como simulador** tridimensional de trabajo para el proyecto.

### 3.6. Alternativas dentro de Naoqi

Una vez que hemos descartado la utilización del software de creación automatizada de comportamientos proporcionado por Aldebaran, *Choreographe*, se toma como punto de partida para la creación del comportamiento complejo el trabajo a un nivel más bajo, interpelando directamente al controlador de Nao, *Naoqi*.

*Naoqi*, como se expuso en el capítulo 2, es el conjunto de software de bajo nivel que interpela directamente al software de Nao. Por ello, el uso de *Naoqi* no parece en principio una elección. Sin embargo, sí lo son los diferentes mecanismos que proporciona *Naoqi* para gestionar las dos tareas básicas que se pueden determinar en el proyecto: adquisición de datos y ejecución de movimientos relacionados con los datos obtenidos.

Las pruebas que se completaron residieron básicamente en la adquisición de los conocimientos indispensables para el manejo del robot, así como de los paradigmas de programación en los que descansa *Naoqi*.

#### Versiones de Naoqi

Durante la elaboración de este proyecto se ha trabajado en todo momento con la versión 1.14.5 de *Naoqi*. Esta es una decisión motivada por la propia contingencia de tener que hacer pruebas experimentales en un robot real. En estos momentos, el robot del que se puede disponer es un Nao cedido por la Universidad de Santiago de Compostela, y la versión de *Naoqi* incluida en el robot es la citada 1.14.5.

#### 3.6.1. Alternativas para el procesamiento de imágenes

Se estudian los distintos mecanismos que nos proporciona la API de *Naoqi* para la obtención de imágenes del entorno de Nao (existirían otros métodos, por ejemplo el ya implementado en el paquete del controlador ROS *nao\_driver*, que nos proporciona un topic (*/camera.raw*) en el que se va publicando la información *cruda* que se obtiene por la cámara). Algunas de las funciones que se estudiarán en este periodo son *getI-*

*mageRemote()*, con la que obtenemos una imagen puntual de la cámara de Nao en el instante de invocación, o *ALVideoDevice.subscribe()*, que nos permite crear un cliente que recibirá los datos de la cámara.

Como es lógico, tras la obtención de los datos, necesitamos un modo de procesado. En el siguiente ejemplo lo que se implementa es únicamente una forma de almacenar de forma local la *captura* de cámara en un instante determinado.

El código-ejemplo presentado es una demostración práctica del procedimiento habitual de captura de imágenes a través del API de Naoqi. Éste podría dividirse en las siguientes fases:

1. Conexión al módulo ALVideoDevice usando un proxy.
2. Creación y suscripción a dicho proxy usando un *cliente de vídeo*
3. Obtención de la imagen usando alguna de las funciones al efecto del API.
4. Cancelación de la suscripción al proxy.
5. Manejo de los datos obtenidos mediante la estructura Image.<sup>6</sup>

```
#_*_ coding: utf-8 _*_
import Image
import os
from naoqi import ALProxy
from naoqi import ALModule
def getImage():
    # Configuramos parametros.
    resolution = 3
    colorSpace = 11
```

---

<sup>6</sup>El contenedor *Image* está constituido por un array que estructuralmente se define como:

[0]: ancho. [1]: alto. [2]: número de capas. [3]: Espacio de color. [4]: *Timestamp (s)*. [5]: *Timestamp (ms)* [6]: Array de tamaño alto \* ancho \* número de capas, que contiene la información de la imagen. [7]: ID de la cámara activa (kTop=0, kBottom=1). [8]: Ángulo izquierdo (radianes). [9]: Ángulo superior (radianes). [10]: Ángulo derecho (radianes). [11]: Ángulo inferior (radianes).

```
fps = 5

# Nos conectamos al proxy VideoDevice de nuestro Nao.
camProxy = ALProxy('ALVideoDevice', '169.254.222.39', 9559)
camProxy.setParam(18, 1)

# Nos suscribimos al proxy, almacenamos la imagen, y nos desinscribimos.
image = None

while (image == None):

    videoClient = camProxy.subscribe('getImage_module',
        resolution, colorSpace, fps)

    image = camProxy.getImageRemote(videoClient)
    camProxy.unsubscribe(videoClient)

# Obtenemos las medidas, y el array de pixeles.

imageWidth = image[0]
imageHeight = image[1]
array = image[6]

# Creamos una imagen a traves de Python Imaging Library (PIL).
im = Image.fromstring('RGB', (imageWidth, imageHeight), array)

# Guardamos imagen en ruta de trabajo del script.

im.save('camImage.png', 'PNG')

# Opcional - visualizamos la imagen usando eog (Eye of Gnome).

os.system('eog camImage.png&')

def main():

    getImage()

if __name__ == '__main__':
    main()
```

## OpenCV

OpenCV es una librería de funciones relacionadas con la visión artificial. Durante el desarrollo del proyecto, se ha tomado en consideración en varios puntos la inclusión de esta tecnología debido a la variedad de soluciones que ofrece, la portabilidad y su avalado funcionamiento a la hora de trabajar con aplicaciones a tiempo real sobre entornos dinámicos.

Se han incorporado varios algoritmos a la hora de detectar colores y formas en el

entorno. Un ejemplo podría ser este código, que detecta rojos en una imagen codificada en el espacio de color HSV, con el que se ha trabajado con el objeto de, en un primer momento, incorporar OpenCV a la arquitectura del proyecto:

```
def getThresholdedImage(imgHSV):  
    imgThresh = cv2.cv.CreateImage(cv2.cv.GetSize(imgHSV),cv2.cv.IPL_DEPTH_8U, 1)  
    cv2.cv.InRangeS(imgHSV, cv2.cv.Scalar(170,160,60), cv2.cv.Scalar(180,256,256), imgThresh)  
    cv2.cv.Smooth(imgThresh, imgThresh, cv2.cv.CV_GAUSSIAN,3,3)  
    return imgThresh
```

Usando algoritmos OpenCV de este tipo, fue posible detectar colores e incluso formas. Por cuestiones de estabilidad y compatibilidad con el entorno, se optó por la utilización del propio Naoqi para la detección de la pelota. Sin embargo y como se verá en el capítulo 10, *Conclusiones y trabajo futuro*, valora en un futuro la implementación de un algoritmo de detección de pelota usando OpenCV.

## Módulo RedBallTracker

Paralelamente al trabajo con OpenCV, se exploró el conjunto de herramientas que proporciona el propio Naoqi para gestionar la detección de objetos a tiempo real. El módulo nativo *redBallTracker* nos permite de forma eficiente, mediante una sencilla API, detectar y seguir una pelota de color rojo con cualquiera de las dos cámaras de Nao. Este módulo está concebido para hacer de puente entre los módulos de detección (agrupados bajo el nombre Naoqi Vision) y los de movimiento (Naoqi Motion).

RedBallTracker hace una llamada a ALTargetDetection, y se queda a la espera de recepción de información. Cuando el objetivo (en nuestro caso, una pelota roja) es detectado, la información de éste se bifurca en dos sentidos: al módulo Motion (para que el movimiento del cuello se adapte dinámicamente a la posición del objetivo) y al módulo Memory (que almacenará en los registros de memoria interna del robot la posición recibida).

Un aspecto crucial que en nuestro caso supone una ventaja con respecto a la posible utilización de las librerías OpenCV para la detección, radica en que la posición del

objetivo se traduce automáticamente a coordenadas relativas a lo que la documentación de Naoqi llama *FRAME\_ROBOT*, y que define como: *referencia espacial que se corresponde con la media de las posiciones de ambos pies de Nao, proyectada sobre el eje cenital Z.*

Con lo cual, al usar *RedBallTracker* estamos dando solución a varios problemas relativos a la adquisición de datos al mismo tiempo. Tenemos:

- Obtención de las coordenadas de la bola de forma dinámica.
- Relación de dichas coordenadas con la línea de referencia FRAME\_ROBOT de Nao.
- Almacenamiento automático de trayectorias en la memoria interna del robot.
- Adecuación de la posición de la cabeza mediante movimiento del cuello para seguimiento de la bola a través del entorno.

## Conclusiones

Por lo expuesto anteriormente, se optó finalmente por trabajar con el módulo RedBallTracker que incorpora el propio Naoqi, en detrimento del uso de las librerías OpenCV. Como se comenta en la sección anterior, el módulo tiene serias limitaciones que afectan a la capacidad de detectar bolas de otros colores y otros tamaños diferentes a los recomendados por Aldebaran. Sin embargo, los puntos a favor que ofrece el uso de RedBallTracker, sobre todo en lo que se refiere a la capacidad de traducir la posición del objeto a detectar a un sistema de referencia espacial sobre la que reside la propia API del robot, resultan claves. Por estabilidad, integración y simplicidad, **se selecciona RedBallTracker para el procesamiento datos de la cámara y detección de la bola.**

### 3.6.2. Ejecución de movimientos

Existen varios módulos del API que nos permiten, con mayor o menor precisión, y con mayor o menor abstracción con respecto a lo que serían las órdenes a bajo nivel

sobre el robot, hacer que éste se mueva. Todas estas alternativas están incluidas en el módulo ALMotion de Naoqi.

### **Joint Control**

*Joint Control* es el conjunto de funciones que permiten manejar directamente la posición de las extremidades (en inglés *joint*) de Nao. Cada uno de los *joints* puede ser manejado de forma independiente, o en paralelo con otros *joints*. Una orden directa el robot para que éste mueva, por ejemplo, el cuello, con el objeto de adecuar la cámara al objetivo (algo útil en el contexto de este proyecto), podría, en Python, tener la forma:

```
# Movimiento simple de cuello, al 10% de la velocidad máxima
names = "HeadYaw"
angles = 30.0*almath.TO_RAD
fractionMaxSpeed = 0.1
motionProxy.setAngles(names, angles, fractionMaxSpeed)
```

*Joint Control*, como se ha dicho, es capaz de definir movimientos para diferentes articulaciones en el mismo instante, y, además, es posible definir trayectorias. Esto quiere decir que podemos definir tiempos, en los que una o varias articulaciones deberían estar en un determinado punto del espacio, en un instante concreto. Un ejemplo básico de esto, es el siguiente fragmento de código (en Python), que mueve el cuello del Nao de forma que la cabeza gira en torno al eje cenital Z y al mismo tiempo lo hace sobre el eje horizontal Y, definiendo una trayectoria en el tiempo:

```
# Definición de dos trayectorias definidas sobre articulaciones diferentes,
# en un mismo lapso de tiempo, con una sola llamada a la función de movimiento.
names = ["HeadYaw", "HeadPitch"]
angleLists = [[[1.0, -1.0, 1.0, -1.0], [-1.0]]]
times = [[1.0, 2.0, 3.0, 4.0], [5.0]]
isAbsolute = True
motionProxy.angleInterpolation(names, angleLists, times, isAbsolute)
```

### Cartesian Control

Constituye el módulo de control de movimientos con el que hemos trabajado. Para la experimentación con las distintas posibilidades de movimientos del robot, y dado el gran número de articulaciones que posee Nao, se ha trabajado con órdenes sencillas y directas, que, en un principio, no deberían comprometer la estabilidad del robot.

En el siguiente código se puede ver el proceso básico de implementación de un movimiento sencillo, en concreto del movimiento de uno de los brazos de Nao, tomando como partida una posición relativa al torso, cuyas coordenadas cartesianas serán (0.0,0.0,0.0) y fijando unas coordenadas-objetivo. El módulo Cartesian Control, se compromete a evaluar la posibilidad del movimiento, y si éste es posible en el estado actual de las articulaciones del robot, lo ejecuta.

Sin embargo, como hemos podido comprobar en la simulación con el robot real, de la que se discute más en detalle en el capítulo 9, *Pruebas experimentales*, se ha comprobado que el robot sí compromete su estabilidad al usar Cartesian Control.

En el ejemplo podemos ver el sencillo proceso de movimiento de uno de los brazos usando Cartesian Control, cuyas fases podrían detallarse como:

1. Conexión al módulo ALMotion usando un proxy.
2. Inicialización articular y motora del robot.
3. Definición de los elementos afectados.
4. Definición del movimiento en etapas intermedias en intervalos temporales.
5. Ejecución del movimiento mediante la función *ALMotion.positionInterpolation*.

```
# Creamos proxy a AMotion
try:
    motionProxy = ALProxy("ALMotion", robotIP, 9559)
except Exception, e:
```

```

        print "No es posible crear un proxy a ALMotion"
        print "Error: ", e

    # Inicializamos motores del robot
    StiffnessOn(motionProxy)

    effector    = "Head"
    space       = motion.FRAME_ROBOT
    axisMask    = almath.AXIS_MASK_VEL      # just control position
    isAbsolute  = False

    # Posicion inicial, que debe ser 0 ya que estamos midiendo de
    # forma relativa
    currentPos = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

    # Definimos los cambios en el tiempo, hasta la posicion final
    dx          = 0.03      # translacion en el eje X (metros)
    dy          = 0.03      # translacion en el eje Y (metros)
    dz          = 0.00      # translacion en el eje Z (metros)
    dwx         = 0.00      # rotacion en el eje X (radianes)
    dwy         = 0.00      # rotacion en el eje Y (radianes)
    dwz         = 0.00      # rotacion en el eje Z (radianes)
    targetPos   = [dx, dy, dz, dwx, dwy, dwz]

    # Definimos el path
    path        = [targetPos, currentPos]
    times       = [2.0, 4.0] # segundos

    # Damos la orden para efectuar el movimiento
    motionProxy.positionInterpolation(effector, space, path,
                                       axisMask, times, isAbsolute)

```

## Conclusiones

Después de la implementación de estos ejemplos descritos y otros que nos permitían mover diversas articulaciones del robot (inclinaciones de torso, movimientos con las piernas, etc.) se concluye que Cartesian Control es una forma altamente precisa, estable e intuitiva de generar movimientos de forma estudiada en el robot. Cartesian Control permite la distribución de movimientos simples y complejos en diferentes fases temporales, de forma aislada o conjunta, y mantiene un control constante sobre la es-

tabilidad del robot y la viabilidad de los movimientos solicitados por el programador a tiempo real. Destaca como principal característica de interés la posibilidad de controlar el movimiento complejo de golpeo de balón diviéndolo en diferentes etapas temporales, e incluyendo de forma separada diferentes patrones para cada una de las articulaciones.

La principal diferencia apreciada entre el uso de Cartesian Control y el de Joint Control es que **Cartesian Control incorpora un módulo de evaluación cinemática inversa**. Esto quiere decir que Cartesian Control, a diferencia de Joint Control, hace un cálculo previo al movimiento. Evalúa la posición de las articulaciones del robot, y en base a ese estado previo, simula la aplicación de la trayectoria requerida para cada una de las articulaciones. Si se determina que la posición y las trayectorias requeridas son incompatibles (en el sentido de que puedan generar inestabilidad en el robot), el movimiento queda invalidado y no se ejecuta.

Podría verse por tanto a Cartesian Control como una capa de refinamiento de Joint Control, con énfasis en la seguridad (por ejemplo, frente a caídas accidentales) del propio robot a la hora de ejecutar los movimientos requeridos por el programador. Es por ello que **se toma Cartesian Control como tecnología elegida** a la hora de implementar los movimientos necesarios para la realización de este proyecto.



## Capítulo 4

# Metodología

En este capítulo se detalla la metodología aplicada. Antes de proponer cualquier diseño de software, se ha tenido en cuenta qué metodología seguir para un mejor acondicionado al contexto de trabajo. Se han barajado varias alternativas de desarrollo de software empresarial y otras más orientadas a proyectos de tipo académico, más abiertas y más indicadas a la variación de las características de los proyectos en función del tiempo.

### 4.1. Metodología ágil iterativa-incremental

Como es lógico dada la naturaleza cambiante y en cierta medida expansiva del proyecto, se optado por una metodología que podría incluirse en el grupo de las llamadas *metodologías ágiles*, con una vocación **iterativa e incremental**. Podría decirse que la metodología aplicada es heredera de la clásica *metodología en cascada* de software, pero incidiendo en el aspecto de que necesitamos acomodar el desarrollo a la constante especificación de nuevas metas, así como a la modificación constante del análisis de requerimientos basados en las pruebas experimentales tanto en simulador como en robot real.

La figura 4.1 ilustra el flujo de fases por las que pasa el proceso de diseño. En las siguientes secciones se detallará cada una de ellas.

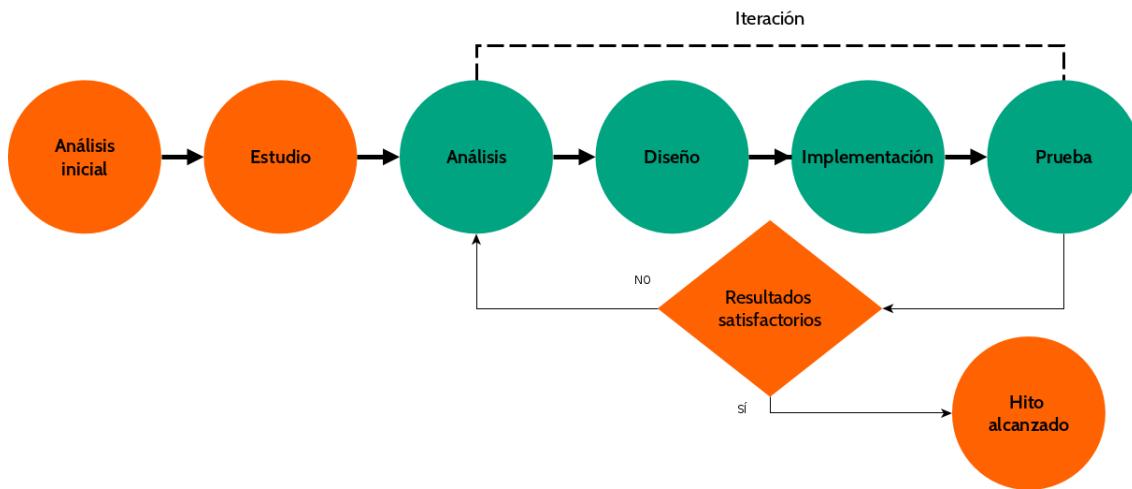


Figura 4.1: Flujo de fases presentes en la metodología seguida

## 4.2. Fase de análisis inicial

En esta fase se plantean una serie de tareas básicas o funcionalidades a cubrir durante el desarrollo, de forma totalmente independiente a la arquitectura que vaya a emplearse, así como de las herramientas que vayamos a utilizar para la implementación.

La elección de las funcionalidades responde a varios parámetros que han de constituir los objetivos del desarrollo: **extensibilidad, reutilización, eficiencia y simplificación** de comportamientos. Esto quiere decir que cada uno de los objetivos debe constituir un comportamiento ampliable, integrable en rutinas más complejas, eficiente en términos físicos/computacionales y simple en el sentido de consistente con las recomendaciones del fabricante.

## 4.3. Fase de estudio de la tecnología

Durante esta etapa se estudiarán y probarán experimentalmente diversas tecnologías, y se decidirá en base de multitud de criterios cuáles de las opciones son las más indicadas para llevar a cabo el proyecto. Cuando las decisiones se hayan tomado, después de los estudios preliminares y la comprobación de la viabilidad/integración conjunta de las tecnologías escogidas, comenzará un proceso de aprendizaje en lo refe-

rido a cada una de ellas. Todo este proceso se ilustra de forma extensa y precisa en el capítulo 3, *Valoración de hardware y software*.

## 4.4. Iteraciones

Por cada iteración se establece un análisis de objetivos. Por un lado tenemos los resultados teóricos esperados, que hemos definido en el análisis de comportamiento previo, y por otro lado tendríamos los resultados que hemos obtenido tras la implementación de las medidas proyectadas. Del análisis de las divergencias entre ambos conjuntos se puede estimar el éxito de la implementación, y se establece la necesidad o no de una nueva iteración, así como los aspectos a tener en cuenta en relación a ella.

Así, englobadas por una iteración, podríamos definir las siguientes etapas lógicas: análisis, diseño, implementación y prueba.

### 4.4.1. Fase de análisis

El análisis de cada una de las iteraciones se distingue de lo que hemos denominado *análisis inicial* en que mientras que éste establecía el conjunto de funcionalidades que el proyecto debería o desearía cumplir, este análisis se centra en una de estas funcionalidades y delimita cómo alcanzar dicha funcionalidad. Cada una de las iteraciones y los resultados de ésta, obligan a repensar la estrategia de aproximación, es por ello que el análisis aparece de manera recurrente en cada una de las iteraciones metodológicas.

### 4.4.2. Fase de diseño

El objeto de esta base es alumbrar el conjunto de soluciones que darían respuesta al análisis efectuado en la fase anterior. Se prestará especial atención a las relaciones entre las distintas entidades que conforman el proyecto, y la forma de interactuación de éstas. Es en esta fase en donde se establece de forma conceptual la arquitectura del software a implementar, así como el dibujo del flujo de datos que sería necesario para la comunicación entre los distintos componentes de la arquitectura. Al tratarse de una arquitectura pensada para la comunicación entre entidades a tiempo real, cabe destacar

que es necesario prestar gran atención a las cadencias de las comunicaciones, así como a las relaciones críticas que se establecen entre las diferentes partes en momentos críticos de los comportamientos propuestos en el análisis inicial.

#### **4.4.3. Fase de implementación**

Una vez efectuado el estudio de los requerimientos que sería deseable cubrir durante la iteración, así como de la arquitectura del diseño que se propone para cubrirlos, comienza la fase de implementación. Esta parte, a pesar de tener un cariz puramente mecánico -siempre y cuando el diseño haya sido propuesto con la suficiente rigurosidad-, podría demorarse más de lo que podría pensarse a priori debido al desconocimiento por parte del programador de muchas de las tecnologías que se han ido incorporando al proyecto. Durante la elaboración del código en las diferentes iteraciones se pudieron observar comportamientos inesperados o erróneos, en muchos casos debido a una implementación en principio incorrecta que requirió de cierta curva de aprendizaje para convertirse en correcta.

#### **4.4.4. Fase de prueba**

La fase de prueba está dividida en diferentes hitos, que, a diferencia de las funcionalidades básicas, engloban un cierto comportamiento que el robot debería poder satisfacer. Así, todas las pruebas experimentales van en relación a ello. Se medirá la fiabilidad de la arquitectura para dar respuesta a un hito, de forma numérica midiendo el error cuando sea posible, y de forma estadística midiendo el coeficiente de acierto cuando el hito tenga solamente dos posibles resultados: positivo o negativo.

Además, esta parte podría dividirse a su vez en dos entornos de prueba diferenciados: simulador, y robot real en entorno real. Durante la mayor parte del proyecto y por razones de logísticas y de eficiencia, se ha tenido que optar por efectuar todas las pruebas experimentales en el simulador 3D que se ha usado finalmente para la elaboración del proyecto (*Webots*, ver sección 3.4.3). La medición de los resultados obtenidos se mide experimentalmente desde varios escenarios iniciales, consiguiendo así diferentes medidas

de eficiencia del sistema en base a la aproximación al resultado que se había establecido en la fase de análisis. Para la contraposición entre resultados y objetivos, se emplean tablas numéricas que miden con precisión las desviaciones.



# **Capítulo 5**

## **Análisis**

En este capítulo se detallará el proceso de selección de funcionalidades de las que pretendemos dotar a nuestro proyecto. Se ha subdividido en un primer apartado, en el que se definen las tareas o requisitos básicos que se debería tener en cuenta a la hora de definir objetivos concretos, un segundo apartado, en el que hace ya un análisis más exhaustivo de ciertas funcionalidades que debería tener el proyecto, y un tercer apartado, que enumera las metas finales.

### **5.1. Análisis de requisitos básicos**

El análisis de requerimientos o tareas a cubrir en el proyecto fue condicionado desde un primer momento por el cariz abierto de la premisa inicial: el desarrollo de comportamientos básicos para el robot humanoide Nao.

Desde este punto de vista, se establece una fase de análisis general del proyecto, de las capacidades del software y hardware con el que se va a trabajar, y, teniendo en cuenta también la bibliografía de la que se dispone y los trabajos previos que se han estudiado, se toma como punto de partida el diseño e implementación de una serie de funcionalidades básicas.

La principal tarea que arroja el análisis inicial del proyecto es la detección de algún

tipo de objeto móvil en el entorno, y la ejecución de alguna clase de movimiento cuantificado y dirigido por los datos obtenidos de esa detección. Así, entre las funcionalidades básicas que se pudieron dibujar en un primer momento, podemos discernir:

- Detección de un determinado objeto en el espacio adyacente al robot.
- Modificación de la posición del robot en relación al objeto detectado.
- Realización de algún tipo de tarea que conlleve la interacción con el objeto.

## 5.2. Análisis de tareas funcionales

Un segundo análisis obligó ya a determinar de forma más precisa ciertos aspectos: qué objetos podemos detectar, cómo podemos hacer que el robot reaccione ante dicha detección, y qué interacción podemos diseñar para con el objeto detectado.

Así, tras esta segunda exploración de funcionalidades, se decide, de forma más concreta, algunas tareas que nuestra arquitectura debería cubrir, en base al análisis principal:

1. Detección por medio de las cámaras frontal e inferior del robot, de una pelota.
2. Capacidad de mover el robot por el entorno de forma estable.
3. Capacidad de adecuar la posición del robot a la posición de la pelota detectada.
4. Capacidad para ejecutar una patada que golpee la pelota sin comprometer la estabilidad del robot.
5. Capacidad de adaptar esta patada a la posición y a otros factores del entorno.

Cada uno de los objetivos (o funcionalidades) es a su vez, en la mayoría de los casos, contenedor del inmediatamente anterior. Esto quiere decir que si conseguimos que, por ejemplo, el robot pueda seguir una pelota, desplazándose a través del entorno, habremos tenido que conseguir previamente que el robot pueda desplazarse -andar, en

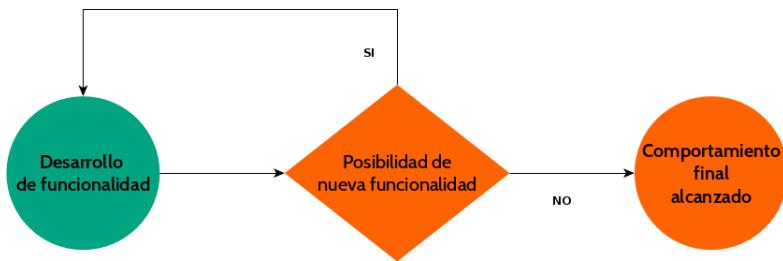


Figura 5.1: Estructura de desarrollo del comportamiento final.

el caso de un robot bípedo- de manera estable y controlada.

Como se puede observar en la figura 5.1, el desarrollo del proyecto puede considerarse una progresión lineal que va cubriendo funcionalidades, y a su vez asumiendo nuevas funcionalidades que integren los anteriores para dotar de una mayor complejidad al comportamiento final. Es decir, el *comportamiento* que se cita en la propuesta inicial del proyecto, pasaría a ser el conjunto de los objetivos individuales, integrados en una sola entidad lógica.

La metodología desarrollo de cada una de las funcionalidades, como se mostró en el capítulo 5, *Metodología*, podría resumirse como se presenta en la figura 5.2:

### 5.3. Objetivos concretos del trabajo

Se establecen, en este tercer nivel, los objetivos concretos, que van a dar lugar a un diseño e implementaciones concretas en los capítulos 7 y 8. A continuación se hace explícita la lista de tareas, con sus respectivas relaciones que darán lugar a este diseño e implementación:

1. **Diseño de un movimiento de golpeo estable**, que no comprometa el equilibrio inercial del robot en ninguna de las fases de golpeo. Además, este módulo debería ser capaz de generar una trayectoria de pelota variable.

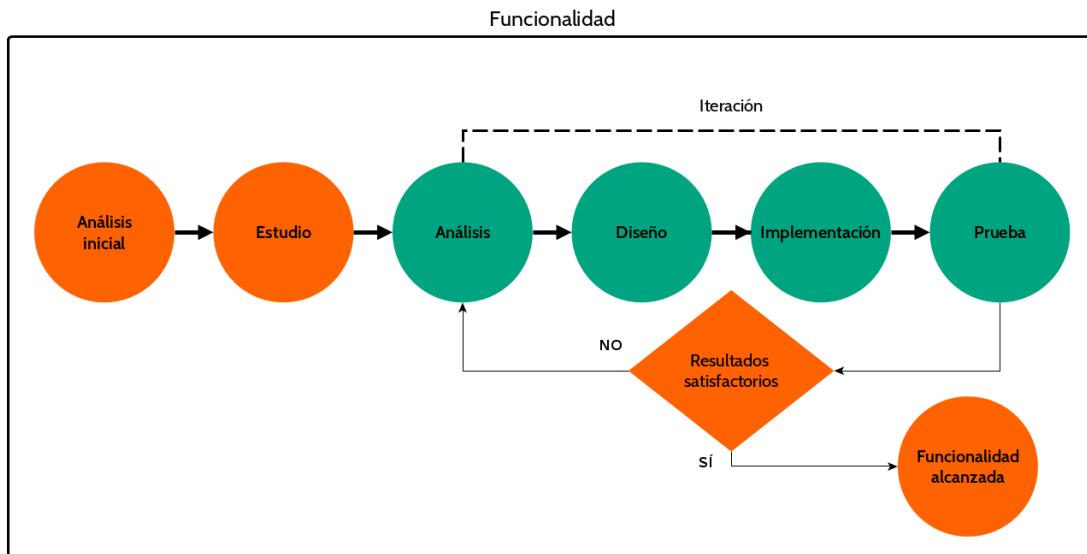


Figura 5.2: Flujo de fases en la creación de una funcionalidad.

2. **Diseño de un detector de pelotas rojas en el entorno**, que incorpore un algoritmo de búsqueda, como el movimiento de cabeza; y que gestione el cambio de cámara activa.
3. **Diseño de algoritmo de acercamiento a la pelota**, que, además de realizar un desplazamiento en base a la posición de la pelota en cada instante, sea capaz de detectar la no-presencia de la pelota en el entorno, enviando así una orden al detector de pelota para que éste cambie la cámara activa.
4. **Diseño de un detector de porterías en el entorno** que, mediante una comunicación directa de la posición de ésta, enviaría al algoritmo de movimiento las órdenes oportunas para que éste moviese al robot buscando una alineación entre los tres elementos que conforman el comportamiento: el robot, la pelota y la portería.

Así, mediante la definición de estos objetivos parciales, se alcanza finalmente la definición de un comportamiento completo, que conforma una unidad lógica, que satisface todos los requisitos básicos planteados en la sección 5.1, y que permite la interacción con humanos: **tirar penaltis**.

# Capítulo 6

## Planificación y costes

### 6.1. Planificación

El proyecto parte de una premisa abierta, y como tal forzosamente tiene que dividirse en al menos dos fases lógicas: el estudio de las **posibilidades**, y el establecimiento de los **objetivos**. Asimismo, podrían definirse dos roles básicos que intervienen en la planificación y que son actores principales de toda la metodología que se ha seguido: el **jefe de proyecto, el analista y el programador**.

Las diferentes etapas en las que se ha dividido la planificación del proyecto se detallan en las siguientes secciones. Son:

1. Estudio de la tecnología.
2. Definición de tareas y análisis inicial.
3. Iteraciones funcionales y pruebas.
4. Pruebas en el robot real.
5. Documentación del proyecto.

### 6.1.1. Estudio de la tecnología

Esta fase, que incluye las decisiones tomadas durante la valoración del software a emplear (y que se documenta ampliamente en el capítulo 3, *Valoración de hardware y software*), se ha dilatado bastante más de lo inicialmente previsto. Debido a la inexperiencia del autor y de las decisiones que éste ha tenido que tomar en una fase temprana de la elaboración, esta fase ha lastrado, como se puede apreciar en la figura 6.1, el desarrollo general de este trabajo.

### 6.1.2. Fase de análisis inicial

Esta fase podría definirse como el estudio previo de los materiales con los que se cuenta a la hora de comenzar el proyecto, así como de las capacidades básicas que ofrece cada uno de ellos, tanto a nivel hardware como software. En este punto, el jefe de proyecto hará una serie de observaciones sobre el material del que se dispone, y será el turno del analista-programador observar cómo podría integrarse ese material para la interacción de los diferentes componentes. Esta fase consta apenas de un par de reuniones informativas, en las que el analista-programador arrojará luz sobre lo que las tecnologías propuestas pueden aportar a la consecución del proyecto, y se cambiarán en caso de que no se amolden a los objetivos, o se detecten incompatibilidades entre ellas.

### 6.1.3. Iteraciones funcionales y pruebas

La implementación de las funcionalidades que se establecieron durante el análisis inicial, es la fase que ha ocupado la mayor parte del proyecto. Se define como la etapa en la que el jefe de proyecto propone diferentes hitos a cubrir, y, en base a los diferentes resultados obtenidos por el analista-programador en el diseño e implementación de éstos, hace una extensión de los objetivos, valora el reducir aspectos de éstos, cambia los parámetros en los que se define un objetivo, etc. Esta forma de proceder se corresponde con lo anteriormente descrito en el capítulo 4, *Metodología*. Esta fase se ha prolongado durante todo el proceso de elaboración del proyecto, ya que la constante

exploración que constituye el método, obliga a hacer ciertas modificaciones de forma también constante en los objetivos.

#### 6.1.4. Pruebas con el robot real

Para las pruebas con el robot real, ha sido obligatorio, por circunstancias externas -la no-disponibilidad del robot- el aplazamiento de esta fase hasta el mes de Septiembre de 2014. Esto ha tenido como consecuencia que las revisiones del diseño y la implementación con respecto a estas simulaciones reales no estén completamente optimizadas. De ello se hablará más en profundidad en el capítulo 9, *Pruebas experimentales*.

#### 6.1.5. Diagrama de planificación

En la figura 6.1 se expone el diagrama de Gantt que explicita la gestión temporal del proyecto. Como puede apreciarse, el estudio de la tecnología sobre la que se ha constituido el proyecto, debido a la inexperiencia del autor, se ha dilatado en gran medida. Conforma, de hecho la mayor de las fases en la planificación, extendiéndose durante un tiempo superior a tres meses. En esta fase se incluye todo el conjunto de pruebas que se ha explicado en el capítulo 3, y que sirvieron para delimitar las tecnologías que se iban a usar y las que se deberían descartar por incompatibilidades con otras partes del proyecto.

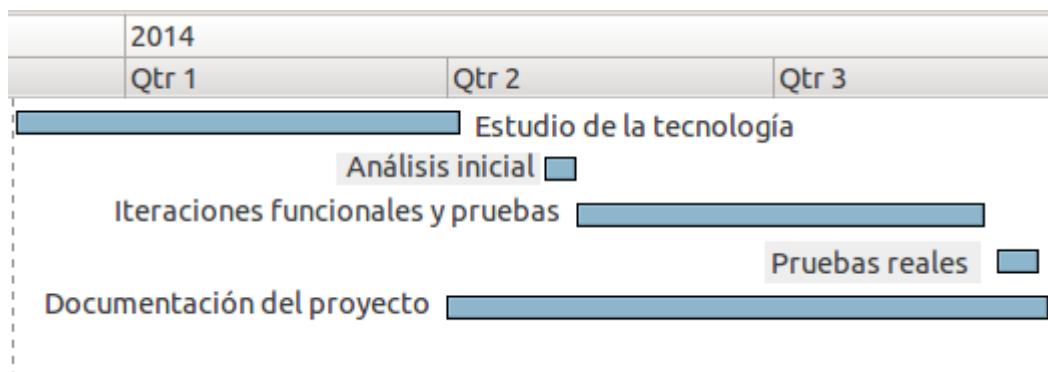


Figura 6.1: Diagrama de Gantt aproximado sobre la gestión del proyecto.

También puede apreciarse que la etapa de pruebas experimentales con el robot real, es demasiado corta para un proyecto de esta naturaleza. Esto se debe principalmente a la imposibilidad de disponer del robot durante los meses de verano, y por tanto el margen de ajuste de la implementación a la hora de adaptarla al robot real (que como se verá en el capítulo 9, es necesario) fue escaso.

## 6.2. Costes

En esta sección se hará una exposición de los costes motivados por el proyecto. Se ha dividido la sección en costes asociados a recursos humanos y en coste asociado a los recursos de hardware y software.

### 6.2.1. Recursos humanos

Para hacer una estimación del coste de personal humano, podríamos establecer que el proyecto ha tenido una etapa de elaboración que se ha prolongado durante unos 8 meses. Existen dos roles representados por dos personas en el proyecto: jefe de proyecto y analista-programador. Se establece la abstracción de que el jefe de proyecto tuviese una retribución por hora de 30€, mientras que el analista-programador la tendría de 20€/hora. Estableciendo un tiempo medio de trabajo diario para el analista-programador de 3 horas, y haciendo lo propio en el caso del jefe de proyecto, aproximando a 20 horas totales de supervisión del proyecto, la siguiente tabla mostraría los costes en cuanto a recursos humanos se refiere:

Rol	Horas dedicadas	€/hora	Coste
Jefe de proyecto	20	30	600
Analista-programador	528	20	10560

### 6.2.2. Hardware y software

En la relación de hardware de la tabla 6.1, se expone el coste total de hardware que ha supuesto el proyecto. El robot Nao tiene un valor monetario asociado nulo debido

### 6.2.3. Hardware

Hardware	Coste
PC portátil Sony Vaio VPCF22	300€
Nao Robot	0€

Tabla 6.1: Costes asociados al hardware empleado en el proyecto.

Software	Tipo de licencia	Coste
ROS Indigo	GNU Public License	0€
Webots EDU 7.4.3	Propietaria	44.07€
NAOqi 1.14.3	Propietaria	0€
Python 2.7.6	GPL-compatible	0€
GCC compiler	GNU Public License	0€
Sublime Text 2.0.2	Propietaria	0€
yEd Graph Editor 3.12.2	Propietaria	0€
pdfTeX	GNU Public License	0€

Tabla 6.2: Costes asociados al software empleado en el proyecto.

a que ha sido una donación en forma de préstamo al proyecto. El valor asociado al PC portátil se calcula como la parte proporcional del tiempo que se ha usado el recurso, con respecto al tiempo real de amortización del mismo.

La mayoría del software empleado no supuso ningún coste adicional para el proyecto. La mayoría del software es libre y gratuito, exceptuando el simulador 3D empleado y el editor de texto. En la tabla 6.2 se puede ver que el único software por el que se ha tenido que desembolsar una cantidad de dinero es Webots, cuya licencia ha sido donada al proyecto y podrá ser usada en futuras aplicaciones, con lo cual, se calcula la parte proporcional suponiendo un tiempo de amortización de 5 años.

### 6.3. Resultados de la planificación

La planificación inicial resultó ser bastante acertada, exceptuando la excesiva duración de la etapa de aprendizaje de cada una de las tecnologías, así como la exploración de éstas y la toma de decisiones asociada. Esto último, lastró el proyecto considerablemente en el tiempo, estableciendo un incremento del tiempo necesario para completarlo de unos 4 meses. Si extrapolamos esta cantidad de tiempo, teniendo el cuenta los factores que hemos descrito en el apartado de costes de recursos humanos, concluimos que el proyecto podría haber sido resuelto con un ahorro de 5280€.

La final prolongación de esa etapa, unida a la poca disponibilidad horaria del analista-programador (estudiante) debida la conjunción de este proyecto con su jornada laboral, hizo que se produjese el citado aumento del plazo previsto, por lo que la finalización y entrega del proyecto tuvo que posponerse hasta el mes de Septiembre de 2014, cuando había sido proyectada su entrega para el mes de Junio de ese mismo año.

# Capítulo 7

## Diseño

En este capítulo se introduce el diseño final del software. En la primera de las secciones se muestra y comenta la arquitectura global del proyecto, mientras que en cada una de las siguientes se expone el diseño de cada uno de los módulos lógicos que dan respuesta a los requisitos finales y concretos del software definidos en el capítulo 5, *Análisis*.

### 7.1. Diseño de funcionalidades

Una de las premisas generales del proyecto desde un primer momento, es la construcción del sistema en base a una arquitectura embebida en ROS. Así, todos los componentes de la arquitectura del proyecto son a su vez componentes de ROS, y por tanto juegan alguno de los roles ROS: nodos, mensajes, servicios, topics, etc. La utilización de ROS como arquitectura general del proyecto además es un aspecto que ha quedado demostrado como efectivo para este proyecto en la sección 3.2.1.

El principal componente de una arquitectura ROS, y por tanto del proyecto es el **nodo**. El sistema constará de un conjunto de nodos que realizan diferentes funciones y se comunican entre sí a tiempo real, mediante topics y sistemas de distribución ya implementados en ROS.

Con el objeto de dar respuesta a las tareas o funcionalidades que se propusieron durante el análisis, se ha propuesto una arquitectura global que descansa sobre ROS y utiliza las herramientas de comunicación y de manejo de parámetros entre los diferentes nodos que ROS proporciona.

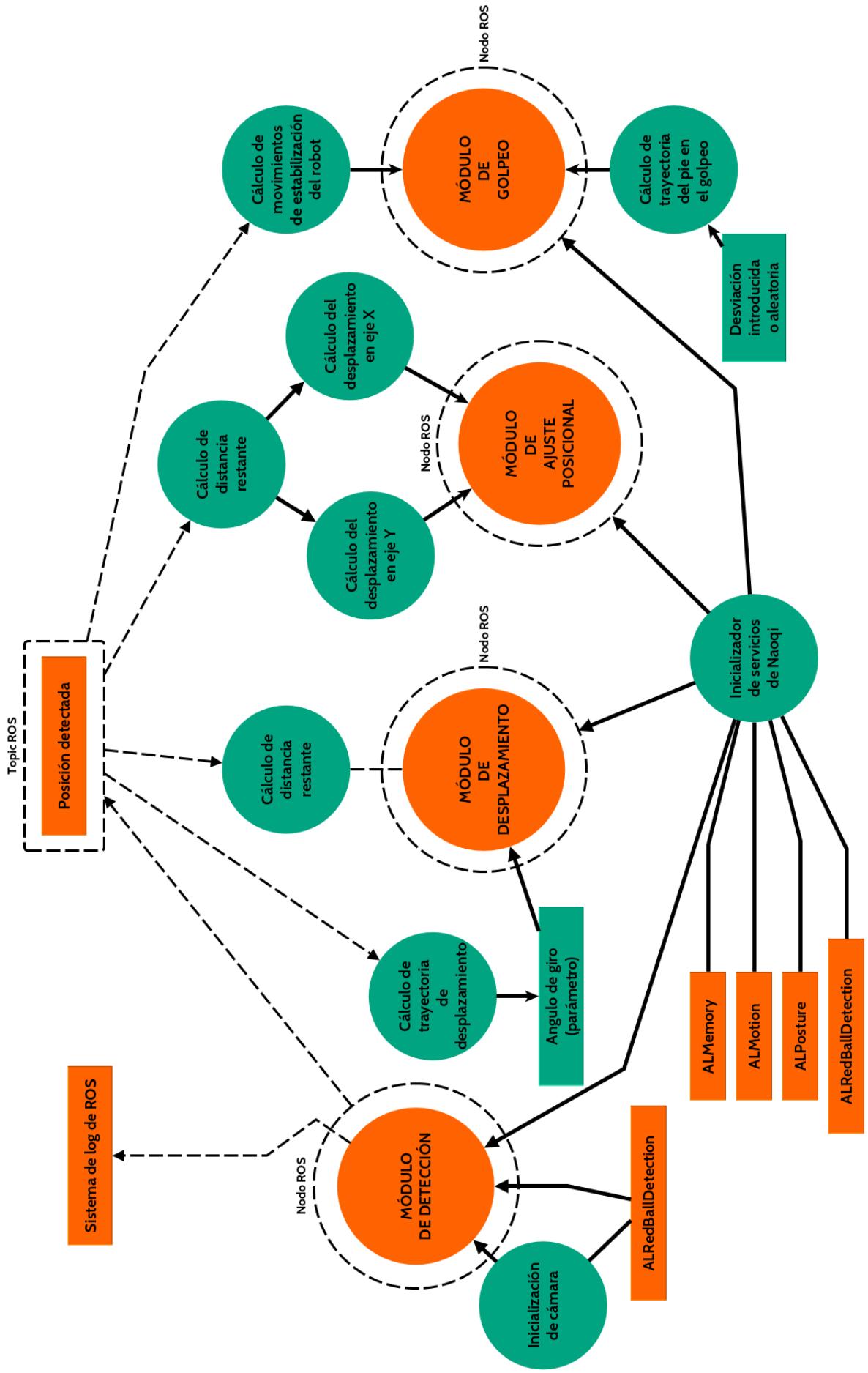
Para el diseño de esta arquitectura global, se han ido realizando una serie de diseños intermedios, que dan respuesta a cada una de las funcionalidades que se propusieron en el análisis inicial. Como ya se ha comentado en el capítulo *Metodología*, cada uno de los diseños llevará asociado una implementación y una serie de pruebas experimentales (realizadas en el simulador 3D elegido, Webots), que a su vez arrojarán unos resultados sobre los que se modificará el diseño implementación.

En esta sección se detalla cada uno de los diseños que se han considerado finales para cada una de las funcionalidades, es decir, se presenta el resultado de las iteraciones que se definen en el capítulo *Metodología*.

## 7.2. Arquitectura global del software

La arquitectura global del software integra las diferentes funcionalidades que hemos ido propuesto, de tal forma que se conforma un sistema estable y distribuido, conectado a través de la red ROS, tal y como se apuntaba en los requerimientos generales del proyecto.

En el diseño, se han dejado como líneas de puntos aquellos componentes y comunicaciones que dependen directamente de ROS y se gestionan mediante dicho sistema. Se han dejado con línea continua las comunicaciones que dependen de Naoqi. El diseño de la arquitectura global puede verse en la figura 7.7.



### 7.2.1. Detección de la pelota

La funcionalidad de detección de pelota se implementa haciendo a su vez uso de ciertos servicios de Naoqi. Como ya se ha comentado en el capítulo *Valoración de hardware y software*, se ha elegido el módulo nativo de Naoqi RedBallDetection para la detección de la bola. Además, para la detección, es necesario utilizar otros módulos de Naoqi, ya que necesitamos que el robot siga con la cabeza el movimiento de la pelota, y almacene internamente la posición actual de la pelota. Para ello necesitamos los módulos de movimiento, gestión postural y gestión de memoria interna que Naoqi incorpora. Además tenemos que gestionar qué cámara está usando el robot en ese momento, usando para ello ciertos parámetros de entrada al módulo. Y en lo que se refiere a la cámara inicializarla, calibrarla, etc., con lo que necesitaremos también una librería que implemente funciones a este respecto.

Por último, tendremos también que encapsular la información en mensajes de ROS, ya que, como vamos a seguir una arquitectura cimentada sobre ROS, el resto de diseños que hagamos tienen que tener una información válida con la que poder trabajar. Por ello, desde un primer momento, el resultado de la computación del módulo de detección, tiene que estar siendo publicada a tiempo real sobre un *topic* ROS.

El diseño, de esta funcionalidad, por tanto, podría pensarse como el que se dibuja en la figura 7.3.

### 7.2.2. Desplazamiento estable

Al igual que la funcionalidad anterior, el diseño de un comportamiento que permita al Nao moverse por el entorno de forma estable, hace uso de ciertos módulos de Naoqi. El más evidente es ALMotion, ya que necesitaremos que el robot se mueva. En cualquier caso, esta funcionalidad se ha diseñado teniendo en cuenta que para la realización del proyecto, además que permitir mover a Nao de forma estable, tenemos que hacer que éste se desplace haciendo giros con un ángulo determinado, ya que será así cómo finalmente se aproxime a la pelota-objetivo. En cualquier caso, un ajuste más fino de

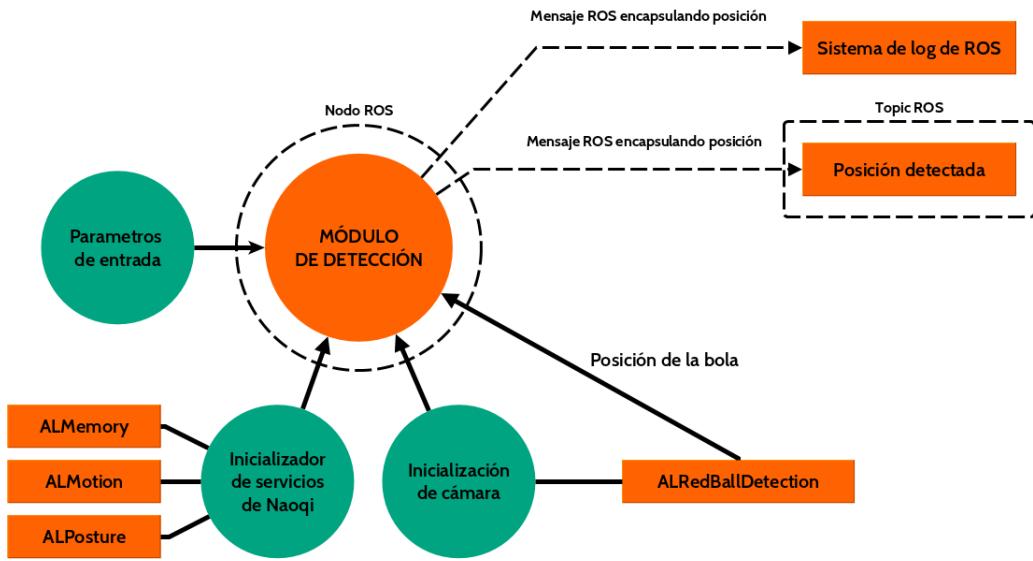


Figura 7.2: Diseño del módulo de detección de posiciones de la pelota.

este comportamiento, se dejará para etapas más avanzadas del desarrollo.

En el presente diseño, se establece una propuesta, para desplazar al Nao con un ángulo variable en su trayectoria, puede verse en la figura 7.3.

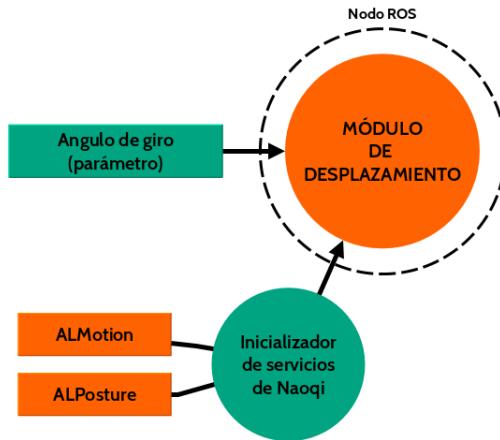


Figura 7.3: Diseño del módulo de desplazamiento estable con trayectoria variable.

### 7.2.3. Desplazamiento adaptativo

Una vez diseñado un comportamiento que permite a Nao moverse por el entorno con seguridad y con un ángulo de giro variable, que permite el trazado de trayectorias diferentes sobre el plano, podemos hablar de un desplazamiento adaptativo, sin más que adecuar ese giro a la posición de un objeto dado.

En nuestro caso, además, por la propia naturaleza del proyecto, podríamos distinguir dos tipos de aproximación a al objetivo. Una primera fase se serviría del comportamiento anterior, desplazamiento con giro, y una segunda fase trabajaría con un ajuste más fino y la cámara inferior, dotando al robot de un mecanismo que permite la colocación en base a la posición relativa del objetivo con una mayor precisión.

Para este punto, se toma como referencia la posición del objetivo que había sido computada y encapsulada en mensaje ROS en la primera de las funcionalidades, la de detección. Así, tendremos que adecuar la posición del robot a los mensajes ROS que están siendo procesados en ese topic a tiempo real.

En este primer diseño, que puede verse en la figura 7.4, se muestra una modificación de la funcionalidad anterior, el desplazamiento con giro, para convertirlo en adaptativo mediante la introducción de un sistema de cálculo de trayectoria de desplazamiento del robot en base a la posición de la pelota que está siendo publicada:

En el segundo diseño, se implementa la funcionalidad de ajuste fino. Este ajuste posicional debería hacerse cuando el robot está en posiciones muy cercanas a la pelota, de tal forma que se adapte su posición con un error estadístico asumible, que debería estar previsto como cota de error del algoritmo de acercamiento. El diseño es similar al anterior. Difiere en que ya no se computa la trayectoria del robot y el ángulo de giro (se asume que el robot está lo suficientemente cercano al objetivo), sino que se establece un módulo de cálculo de la distancia restante al objetivo. Este módulo debería computar las diferencias y calcular un desplazamiento frontal (en sentido del eje X) positivo para acercar al robot al objetivo, o negativo para alejarlo. Para el eje Y debería seguir un

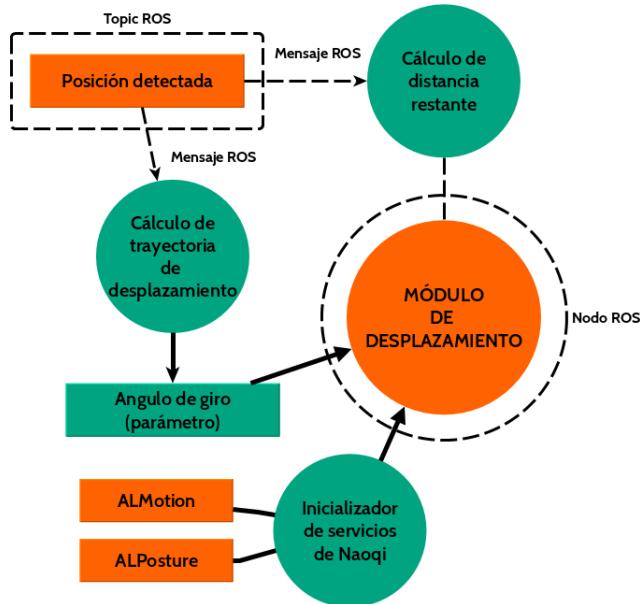


Figura 7.4: Diseño del módulo de desplazamiento adaptativo.

procedimiento análogo, con su respectiva cota de error.

Así pues, el diseño de esta segunda funcionalidad de acercamiento a objetivo, se ha pensado como se muestra en la figura 7.5.

#### 7.2.4. Golpeo estable

Para el diseño de esta funcionalidad, partimos de la abstacción de que el robot y la pelota están alineados en una posición óptima, lo cual estaría en principio garantizado por la funcionalidad de *desplazamiento adaptativo* que hemos diseñado en la fase de diseño inmediatamente anterior. Así, el objetivo de esta fase es diseñar un sistema que permita que el robot extienda la pierna derecha hacia adelante, de tal forma que el pie derecho impacte con la pelota.

Para garantizar que el robot no sufra inestabilidades derivadas del golpeo, ha de buscarse una referencia previa de la posición exacta de la pelota, así como una concatenación de movimientos de preparación para el golpeo acorde con ésta. De esta forma,

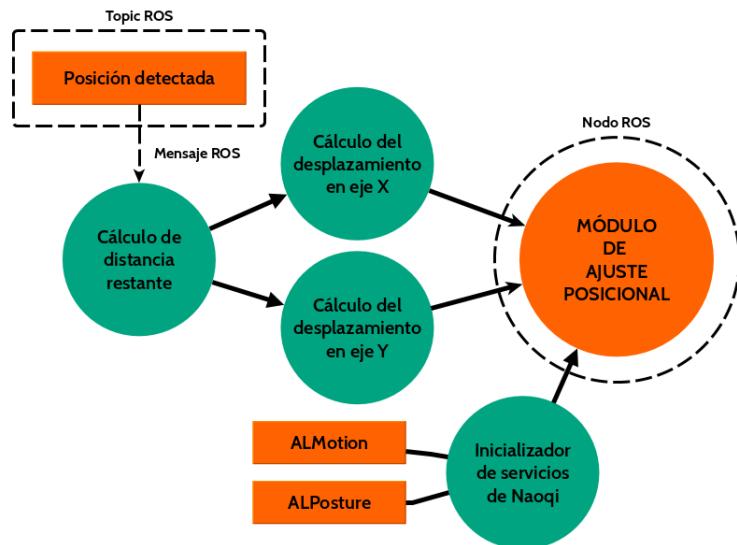


Figura 7.5: Diseño del módulo de ajuste posicional.

encontrando los parámetros adecuados, el golpeo se efectuará sin riesgos.

Como puede apreciarse, el diseño de esta fase es sencillo, no implica demasiada interacción con la arquitectura global, pero es una fase ineludible para funcionalidades futuras que incluyan una entrada de datos con respecto a la dirección de la patada, y una salida de datos en forma de medición de la trayectoria, como corresponderá a la funcionalidad siguiente.

El diseño de una rutina de golpeo de pelota simple podría verse como el que se muestra en la figura 7.5.

### 7.2.5. Golpeo configurable

La funcionalidad de golpeo variable difiere de la de golpeo en el aspecto de interacción con el usuario del software. Cuando se usa el golpeo variable, se entiende que el usuario tendrá que especificar la cantidad de variación en el golpeo, bien sea a través de la introducción de un ángulo en el movimiento, bien de una cierta desviación lineal en cada una de los puntos que conforman la trayectoria predefinida.

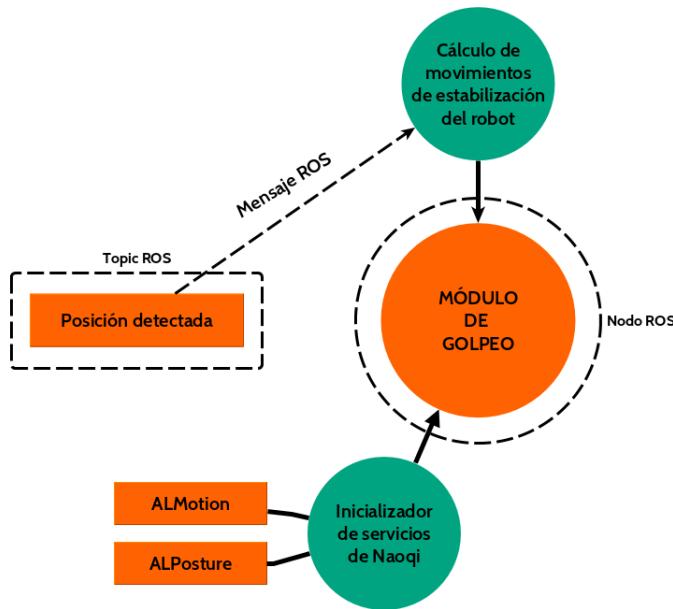


Figura 7.6: Diseño del módulo de golpeo básico.

Además, se podría integrar en esta fase la aleatoriedad de la variación introducida. Es decir, podría pensarse como un sistema que efectuase una patada a la pelota con una variación diferente en cada ejecución, de tal forma que la trayectoria de la pelota fuese en principio desconocida. Así, tendríamos un sistema que obligaría a un posible portero (por ejemplo, un niño) a intentar detener la pelota sin conocer previamente la dirección de ésta. En este punto habríamos conseguido adaptar nuestro sistema para verlo como un posible juguete.

El diseño de esta funcionalidad apenas difiere de la anterior, únicamente se introducen unas variables de entrada, que pueden ser configurables en tiempo de ejecución o aleatorias. El diseño puede apreciarse en la figura 7.7.

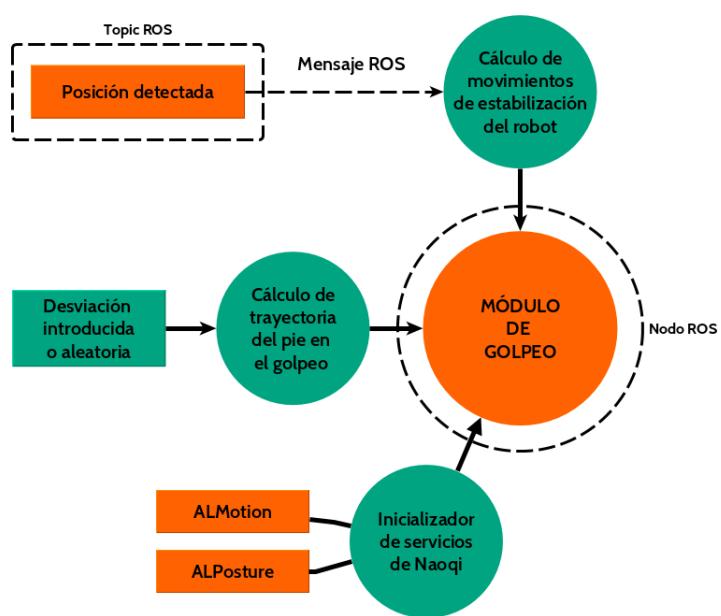


Figura 7.7: Diseño del módulo de golpeo configurable o adaptable.

## Capítulo 8

# Implementación

La implementación del software que se detalla a continuación, pretende dar respuesta a la arquitectura que ha sido presentada en el capítulo anterior. Para ello, se han implementado una serie de nodos, todos ellos escritos en Python, que pretenden, de forma individual, dar respuesta a cada una de las funcionalidades requeridas, y, de forma conjunta, a la arquitectura global diseñada.

Además, siguiendo el paradigma ROS (*ROS-way*), se ha pensado cada uno de los módulos python como nodos ROS, cuyos procesos deben estar presentes, estén realizando comunicaciones activas o no, durante todo el tiempo de ejecución del software. Además de los nodos, y para fomentar la reutilización de código y la incorporación de soluciones a futuros proyectos, se han escrito una serie de librerías de funciones y servicios, también en Python, que complementan y simplifican el funcionamiento de los nodos. Automatizan, por ejemplo, la utilización recurrente de ciertas funciones o ciertos fragmentos de código que han de usarse de forma repetida, ya que son rutinas compartidas (por ejemplo ciertas interacciones con el núcleo de Naoqi, ciertos accesos algo complejos a funciones de Naoqi, etc.).

Los principales nodos y librerías de los que consta el proyecto, con sus características y justificación de existencia pertinente, se exponen en las siguientes secciones. Los módulos librerías que trataremos son:

- Módulo de inicialización (*init\_robot*).
- Módulo de gestión de las cámaras (*camera*).
- Módulo detección (*ball\_pos*).
- Módulo de desplazamiento (*walker*).
- Módulo de ajuste de posición para golpeo (*set\_position*).
- Módulo de golpeo (*kicker*).

### 8.1. Módulo de inicialización (*init\_robot*)

Realiza las gestiones básicas en torno a la inicialización del robot. Gestiona la activación de motores articulares y establece una serie de funciones que mapean los diferentes proxies de Naoqi para el manejo de el movimiento, postura, y módulos de detección del entorno. Funciona como una pequeña librería, se ha optado por implementar un módulo de este tipo, además, debido a la agilidad que proporciona a la hora de trabajar de forma interactiva con el robot a través de, por ejemplo, un intérprete python conectado a Naoqi.

### 8.2. Módulo de gestión de las cámaras (*camera*)

Gestiona la cámara. Su razón de ser es muy semejante a la del módulo *init\_robot*, conforma también una pequeña librería, pero se ha optado porque sea una entidad lógica propia. Realiza funciones de inicialización, y proporciona una interfaz para obtener imágenes. Facilita también el cambio de cámara activa en tiempo de ejecución. Al igual que *init\_robot*, agiliza en gran medida el trabajo interactivo con el robot, por ejemplo para tareas de depuración.

### 8.3. Módulo detección (*ball\_pos*)

Haciendo uso de la interfaz definida en los módulos *init\_robot* y *camera*, el módulo *ball\_pos* obtiene de forma reiterada la posición de la pelota, si es que ésta se encuentra en

el rango *visible* del robot. Procesa dicha información y la empaqueta como un mensaje estándar ROS, mediante la definición de sus coordenadas en relación al sistema de referencia *FRAME\_ROBOT*. En la figura 8.1 puede verse gráficamente la definición de este sistema de referencia espacial. Los mensajes de posicionamiento, así, son enviados desde el nodo ROS al resto de la red. La posición de la pelota es a su vez publicada por este módulo sobre el topic de ROS *pos*, de donde otros nodos del sistema obtendrán la información a tiempo real.

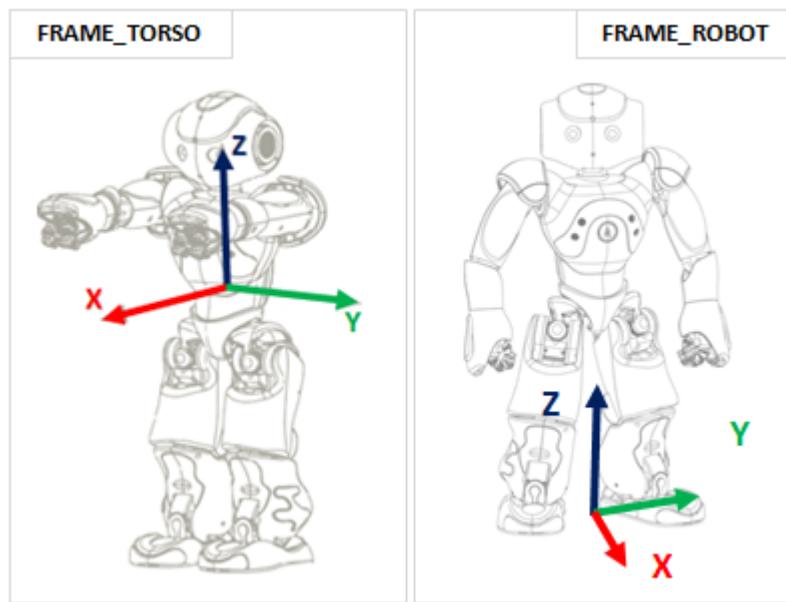


Figura 8.1: Sistema de referencia definido por Naoqi para la implementación de movimientos coordinados.

En las primeras versiones del software, el módulo *ball\_pos* requería un parámetro de entrada: la cámara con la que Nao debía, en ese momento, rastrear el entorno en búsqueda de la pelota. Con las sucesivas depuraciones del sistema, se ha optado por la eliminación de este parámetro. Existe una limitación dada por el propio Naoqi en su versión 1.14.5: el módulo nativo *redBallTracker*, del que hace uso extenso *ball\_pos*, sólo permite el uso de una cámara en un instante determinado de tiempo. La arquitectura, sin embargo, implementa un sistema de detección de pelota propio, de tal forma que Nao, en tiempo de ejecución, cambia automáticamente la cámara asignada al módulo *redBallDetection* si la pelota no está siendo detectada.

El módulo funciona además como un *publisher* clásico de ROS. En el siguiente fragmento de código puede verse la implementación básica de este aspecto:

```
def publish_ball(redBallTrackerProxy, pub):
    ball = Point()
    position = redBallTrackerProxy.getPosition()
    ball.x, ball.y, ball.z = position[0], position[1], position[2]
    pub.publish(ball)
    pub = rospy.Publisher('pos', Point, queue_size=1)
    rate = rospy.Rate(5) #5Hz
    [...]
    publish_ball(redBallTrackerProxy, pub)
```

#### 8.4. Módulo de desplazamiento (walker)

Cuando el módulo *walker* es lanzado, espera a que sean publicados mensajes a través del topic *pos* -en nuestro caso, mensajes de tipo *Point*; coordenadas espaciales que estaría publicando el nodo *ball\_pos*. Cuando un mensaje de posición de la pelota es detectado por el módulo, éste se encarga de mover el robot de forma acorde a la información recibida. El módulo desplaza el robot hacia adelante, de forma regular, y al mismo tiempo establece un ángulo de giro para dicho caminar. Es decir, calcula, en base a las coordenadas de la pelota, el giro que debe de aplicar al movimiento del robot, y por supuesto el sentido de éste, siempre relativo a la posición del propio robot.

Las funciones básicas que implementan el movimiento del robot, una vez se han creado los oportunos proxies a Naoqi mediante el módulo de inicialización, son *calculate\_theta* y *move\_robot*:

```
def calculate_theta(ball):
    theta = math.copysign(math.pi/16, ball.y)
    return theta

def move_robot(motionProxy, ball):
```

```

if not rospy.get_param('move'):
    return
theta = calculate_theta(ball)
motionProxy.moveTo(0.15, 0.0, theta)

```

Es decir, el robot se mueve uniformemente hacia adelante, evaluando la posición relativa de la pelota con respecto a sí mismo cada 15 cm. que recorre, y condicionando a ello el ángulo de giro a tiempo real, que como se observa en el fragmento de código, se calcula de la forma:

$$\theta = \frac{y \cdot \Pi}{16}$$

Cabe destacar que el módulo se encarga además de comunicar la presencia de la pelota en el espectro visible del robot. Este comportamiento ha sido diseñado como un *feedback* que aprovecha las capacidades del Parameter Server de ROS. Si se lanza una excepción cuando el módulo intenta leer el topic *pos* (es decir, no se están publicando posiciones), se activa un flag, implementado como parámetro ROS, que hace que el módulo de gestión de la cámara cambie la cámara activa del robot. Esto puede verse en el siguiente fragmento de código:

```

try:
    rospy.set_param('no_ball', 0)
    ball = rospy.wait_for_message('pos', Point, 2.0)
except rospy.exceptions.ROSEException:
    rospy.set_param('no_ball', 1)

```

## 8.5. Módulo de ajuste de posición para golpeo (set\_position)

El funcionamiento de este módulo es similar al descrito para *walker*. El nodo lee la posición de la pelota, que es publicada por el nodo *ball\_pos* sobre el topic *pos* y hace un ajuste fino de la posición del Nao con respecto a las coordenadas de la pelota que está siendo detectada. Este módulo se activa cuando la posición que se recibe de la pelota está por debajo de un umbral configurable, de tal forma que el ajuste fino

solamente tendrá lugar cuando el robot se encuentra en posiciones próximas a la pelota.

Dado que este ajuste pretende ser más fino, se ha escrito un algoritmo que coloca el robot utilizando pequeños movimientos hasta que la posición de la pelota se haya a una determinada distancia del robot. Tanto la distancia que recorre el robot en cada uno de estos pequeños pasos, la posición objetivo, y el error máximo cometido, son variables fácilmente configurables por el usuario del software. Para las demostraciones y tras diversas pruebas experimentales (que se detallarán en el capítulo posterior) se ha elegido la siguiente configuración:

```
TARGET_X = 0.12  
TARGET_Y = -0.05  
ERROR_X = 0.01  
ERROR_Y = 0.005  
STEP_X = 0.03  
STEP_Y = 0.015
```

De esta forma, cuando el algoritmo converge y la cota de error máximo se supera, éste detiene su ejecución, dejando al robot en una posición idónea para ejercer el golpeo del balón con pierna derecha. Además, el módulo es el encargado de configurar el parámetro ROS que gestiona la ejecución de la patada, activando el flag que hace que el módulo de golpeo comience su ejecución.

## 8.6. Módulo de golpeo (kicker)

El módulo *kicker*, al igual que los anteriores, crea su propio nodo ROS, y lee datos sobre el posicionamiento actual de la pelota, en el caso de que esté siendo detectada, desde el topic *pos*. Este módulo tiene dos formas básicas de funcionamiento: golpeo básico (rectilíneo) o golpeo con dirección variable (y, en este caso, en dirección configurable o aleatoria dentro de un determinado rango).

Hace un uso intensivo de la herramienta Cartesian Control de Naoqi (de cuyas capacidades se habla de forma más extensa en la sección 4.4.3). Este nodo hace una

estimación todavía más exacta de la posición de la bola a tiempo real. El algoritmo asume que la pelota se encuentra inmediatamente delante del pie derecho del robot, en una posición que ha sido ajustada antes por el nodo *set\_position*, y comienza a hacer un muestreo de N tomas -parámetro *frames*, ajustable mediante el Parameter Server de ROS-. Una vez obtenido el muestreo completo, y asumiendo que la pelota no sufrirá ninguna modificación de su posición en este lapso temporal, el nodo estima las distancias al balón mediante la media aritmética de las coordenadas obtenidas. El golpeo de la bola se hace interpolando directamente a Cartesian Control. Para ello, se ha comprobado experimentalmente que es posible modificar la trayectoria de salida de la bola en base a la trayectoria que el pie del Nao describa en el momento de ejecutar toda la secuencia de golpeo.

Antes del golpeo, sin embargo, es necesario que el robot se encuentre en una posición que se estable en sí misma, y además permita la ejecución de la patada, es decir, el movimiento del pie derecho sin superficie de apoyo, unos centímetros por encima del nivel del suelo, sin que el Nao pierda en ningún momento dicha estabilidad. En el figura 6.3 se puede ver al robot en el entorno de simulación Webots adoptando la posición elegida como previa al golpeo.

Se han definido dos puntos a la hora del golpeo, aunque podría haberse definido uno sólo (el punto de golpeo final). El motivo es dotar al movimiento de mayor naturalidad en lo que se refiere a semejanza con el que sería el movimiento natural de una persona que debe golpear una pelota con el pie derecho. Así, tenemos: punto de impulso o **punto A**, y punto de contacto o **punto B**. En las figuras 8.3 y 8.4 se hace una explicación gráfica de cómo se ha definido los puntos.

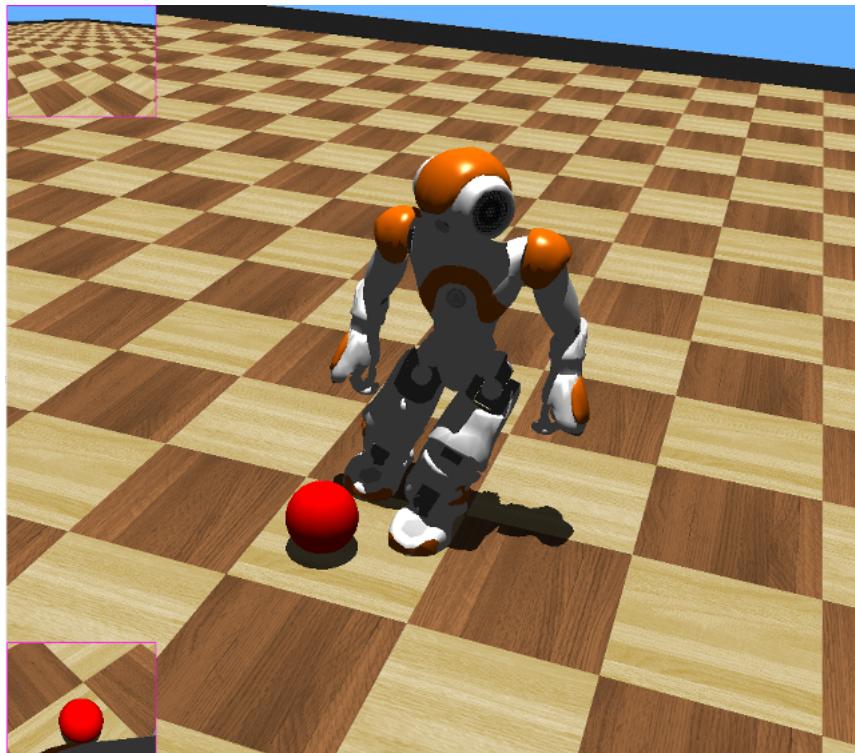


Figura 8.2: Nao en el instante previo al golpeo con pierna derecha.

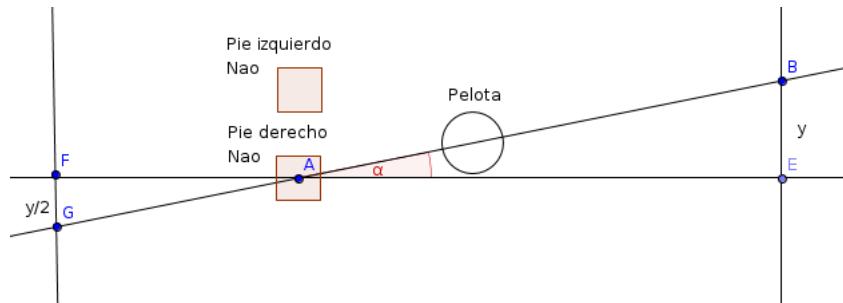


Figura 8.3: Vista cenital de la trayectoria a través del eje Y.

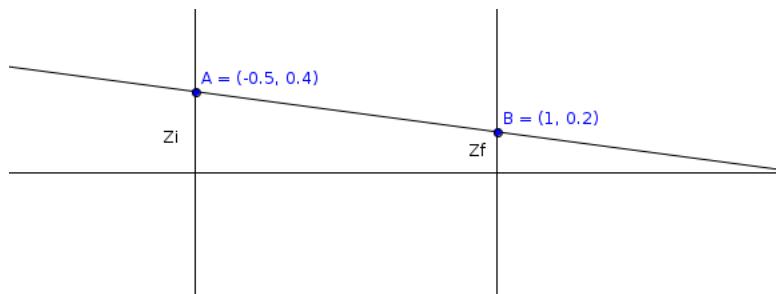


Figura 8.4: Vista lateral de la trayectoria a través del eje Z.

## Capítulo 9

# Pruebas experimentales

En este capítulo se documentan las pruebas que se han hecho, tanto con el robot real como con el simulador Webots, para probar el software que se ha diseñado e implementado. Las pruebas pueden dividirse en dos grandes grupos: pruebas de golpeo y pruebas de posicionamiento. Como ya se ha comentado en el capítulo 6, *Planificación y costes*, las pruebas con el robot real no han sido demasiado extensas en el tiempo por la no-disponibilidad del robot. Sin embargo, éstas han resultado lo suficientemente concluyentes como para proponer mejoras al alguno de los algoritmos, como se verá en este mismo capítulo.

### 9.1. Golpeo con ángulo variable

Definimos *golpeo básico*, como aquel que no establece ninguna variación en cuanto al desplazamiento del pie a través del eje Y. Es decir, no existe variación introducida por el usuario ni obtenida de forma aleatoria por el software en el golpeo. El pie del Nao se desplaza en trayectoria rectilínea paralela a su eje Y relativo. El *golpeo variable*, pues, se define como áquel que incorpora cierto parámetro de desviación con respecto al golpeo simple.

Desviación sobre el eje Y para diferentes ángulos de golpeo			
Ángulo (grados)	Exp.1 (metros)	Exp.2 (metros)	Exp.3 (metros)
36	0.43	0.42	0.45
24	0.29	0.289	0.219
12	0.093	0.12	0.088
0	0.058	0.08	0.095
-12	0.18	0.18	0.165
-24	0.24	0.254	0.214
-36	0.37	0.33	0.37

Tabla 9.1: Desviación sobre el eje Y para diferentes ángulos de golpeo.

### 9.1.1. Objetivo de la prueba

Este experimento se ha llevado a cabo midiendo la desviación de la trayectoria de la pelota cuando sale despedida, con respecto a la desviación introducida en la propia patada. Así, se pretende obtener una relación, y medir la fiabilidad del algoritmo de golpeo con desviación.

### 9.1.2. Resultados en simulación

Como se puede apreciar en la tabla, los resultados en simulación tienen un patrón de repetición bastante ajustado, la desviación media entre los tres experimentos es de 0.0162, lo que nos indica que el resultado de los experimentos en particular para cada ángulo introducido es bastante fiable.

### 9.1.3. Resultados con robot real

En esta ocasión, la tabla muestra una gran divergencia entre el resultado esperado, basado en la simulación, y el obtenido en el robot real. Como se puede apreciar, además, el ángulo de golpeo no se transmite de forma directa a la trayectoria. La desviación en las mediciones de la tabla, en este caso asciende a 0.1442. Un rango de desviación de 14 cm. en un experimento que computa valores de menos de 25 cm. no es aceptable, y

Desviación sobre el eje Y para diferentes ángulos de golpeo		
Ángulo (grados)	Exp.1 (m. aprox.)	Exp.2 (m. aprox.)
25	0.15	0.08
20	0.03	0.12
15	0.08	0.15
0	0.05	-0.20
-15	-0.04	-0.04
-20	-0.05	-0.12
-25	-0.03	-0.10

Tabla 9.2: Desviación sobre el eje Y para diferentes ángulos de golpeo.

refleja la casi nula repetibilidad del experimento.

Dado el resultado del experimento, se introdujo la posibilidad de modificar los parámetros de la patada original para conseguir una mayor precisión y amplitud (que como se puede observar en la tabla anterior, es muy baja con relación a las amplitudes que se consiguen en el sentido opuesto) en los golpesos con ángulo positivo, es decir, tomando el sistema de referencia del propio robot, los que hacen que la trayectoria de la pelota tras el golpeo salga dirigida hacia el lado izquierdo. Para ello, se optó por cambiar durante el experimento la trayectoria de la propia patada, de tal forma que todos los puntos que la definen se mueven algunos centímetros a la derecha, lo cual ocasiona que la pelota sea golpeada también por ese lado, y salga con un ángulo mayor.

En la tabla siguiente pueden verse los resultados de este experimento, se ha trabajado en todas los golpesos con un ángulo de 15 grados, y se ha ido modificando el desplazamiento a la derecha indicado.

Como se puede desprender de la tabla, la modificación no causó gran interferencia en lo que estaba siendo la trayectoria de la pelota cuando esta es golpeada. Cabe destacar

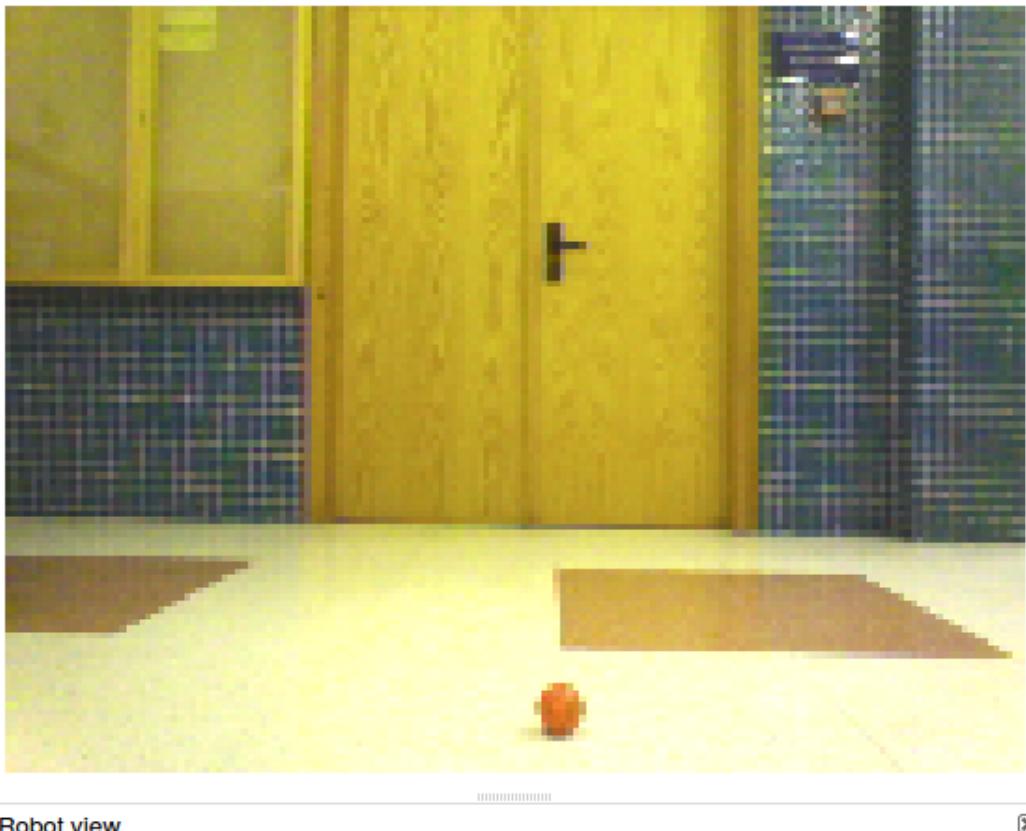


Figura 9.1: Captura de la propia cámara del robot, detectando la bola.

Desviación sobre el eje Y para diferentes ángulos de golpeo	
Ángulo (grados)	Experimento (m)
15° + 2cm. (desviación)	0.10
15° + 3cm . (desviación)	0.10
15° + 4cm. (desviación)	pérdida de equilibrio

Tabla 9.3: Desviación sobre el eje Y para diferentes ángulos de golpeo.

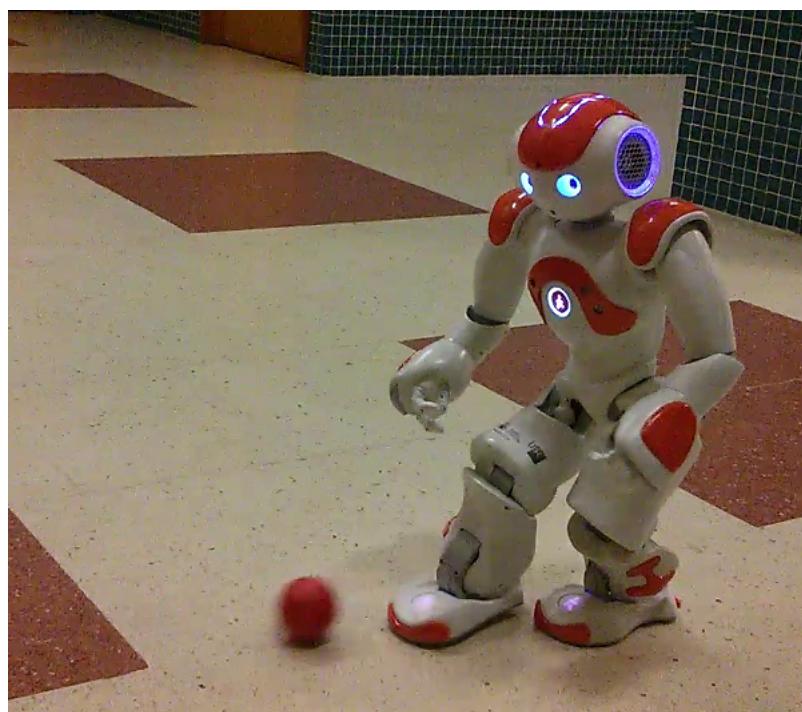


Figura 9.2: Captura de cámara externa, con Nao golpeando la pelota durante el experimento.

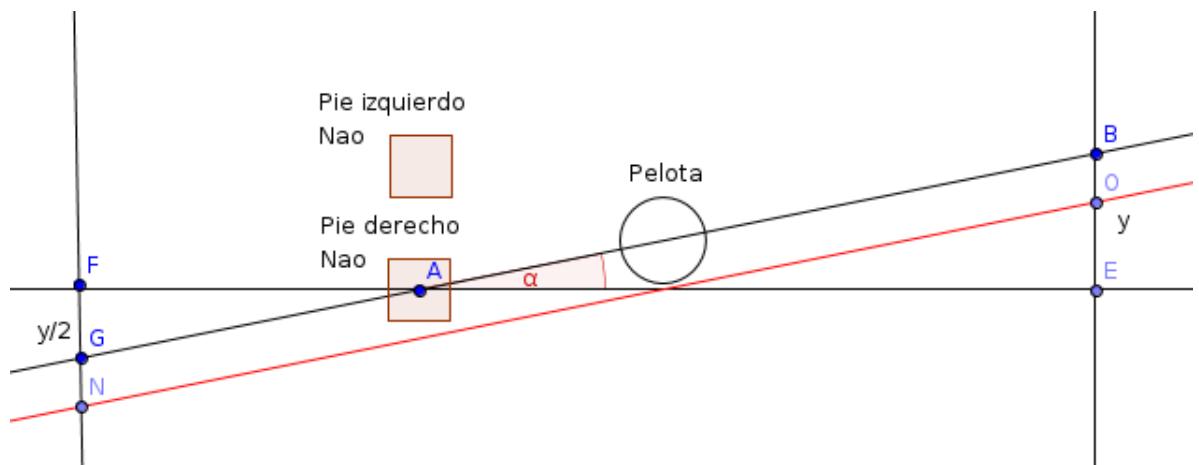


Figura 9.3: Patada modificada respecto al eje Y para el ajuste del golpeo.

Desviación sobre el eje Y,Z para diferentes ángulos de golpeo	
Ángulo (grados)	Experimento (metros aproximados)
15° + 2cm. (desviación) + 1cm. (altura)	0.10
15° + 2cm. (desviación) + 2cm. (altura)	0.15
15° + 2cm. (desviación) + 3cm. (altura)	0.20

Tabla 9.4: esviación sobre el eje Y,Z para diferentes ángulos de golpeo.

que el Nao perdió el equilibrio cuando la trayectoria de la patada se desplazó hasta los 4 cm. con respecto a la original. Así, tras el fracaso de este experimento, se opta por modificar también la trayectoria con respecto al eje Z, es decir, las alturas de los puntos de inicio y finalización de la patada, como puede verse en la figura 9.4:

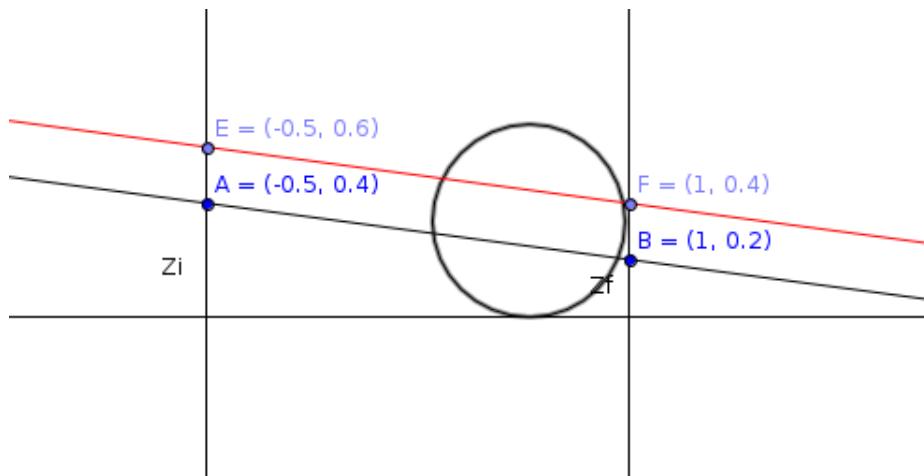


Figura 9.4: Patada modificada respecto a los ejes Y,Z para el ajuste del golpeo.

Los resultados aproximados, tras esta segunda modificación de la trayectoria de la patada, fueron los que se puede observar en la table 9.4.

Con lo que se concluye, con las reservas lógicas dado el reducido número de pruebas que se realizaron, que la modificación de la altura de la patada, en este caso, incide de forma directa y lineal sobre la trayectoria de la pelota seguida. Y además permite optimizar el algoritmo de golpeo que habíamos diseñado, al haberse comprobado una respuesta más ajustada para estos valores más elevados de altura de golpeo.

### 9.1.4. Discusión

Se ha observado en las pruebas experimentales con el robot real, que la posición de la pelota es algo crítico a la hora golpear; determina por completo la trayectoria. La imposibilidad de colocar el robot y la pelota, en un entorno real, en exactamente las mismas posiciones en cada uno de los experimentos, unido a la inestabilidad por el poco peso de la pelota utilizada, hacen que los estados de inercia varíen constantemente y el experimento se vuelva impredecible, contrariamente a lo que ocurría con este mismo experimento en simulación.

A tenor de las modificaciones introducidas en la trayectoria de la patada, en los últimos experimentos, concluimos además que puede introducirse un desvío en cuanto al eje Y de incidencia en la pelota, y también en cuanto al eje Z, es decir, a la altura. Esto nos obliga a cambiar nuestro algoritmo de partida: **tendremos un tipo de golpeo diferente para cada sentido del ángulo de desviación**, aspecto éste que no habíamos previsto a través de las pruebas con simulador, y que las pruebas experimentales con robot real nos han obligado a modificar.

## 9.2. Detección de pelota

### 9.2.1. Objetivo de la prueba

Se pretende cuatificar la precisión de la percepción, por parte del Nao, de una pelota roja. Para ello, se construye un setup experimental en el simulador Webots, consistente en la colocación de la pelota en un punto que pertenece al ángulo de visión del Nao, y después se desplaza ésta hasta el punto en el que queremos comprobar la percepción. Además de anotar si la detección se produce, anotamos el error absoluto y relativo en la interpretación de esta posición por parte del robot.

### 9.2.2. Resultados en simulación

Para esta prueba, se han ido introduciendo pequeñas modificaciones en el simulador en lo que respecta a la posición del balón, y se ha ido comprobando experimentalmente en qué rangos de posiciones el módulo de detección, *ball\_pos*, publica exactamente las mismas modificaciones. Es decir, en qué rango la estimación de la posición del balón es fiable.

Dado que se ha implementado un comportamiento que modifica la cámara activa en cada momento si la pelota no se ha detectado, omitiremos este parámetro en los resultados. Si la pelota se está detectando, en principio, no se especifica con qué cámara se está haciendo, se trata el módulo de detección como una unidad lógica independiente de la implementación y del hardware que se esté usando para la percepción.

Dejamos la coordenada con respecto al eje Y fija, y vamos modificando la posición de la pelota con respecto al eje X. En la tabla se muestra: la posición de la pelota, medida por el propio Webots, la posición interpretada por el Nao, y el error absoluto de esta estimación por parte del robot:

En la tabla 9.5, la posición del balón se ha ido modificando únicamente en torno al eje X, dejando la posición de la bola con respecto al eje Y fija. En este caso, hemos dejado la posición de la bola con respecto al eje Y en el centro del ángulo, es decir, la coordenada Y es cero para todas las variaciones de la coordenada X.

Para la tabla 9.6, la posición del balón se ha ido modificando únicamente en torno al eje Y, dejando la posición de la bola con respecto al eje X fija. En este caso, hemos dejado la posición de la bola con respecto al eje X en la posición óptima según se desprende de la tabla anterior, es decir, la coordenada X es 0.5 para todas las variaciones de la coordenada Y.

Detección de la pelota a través del eje X		
Distancia introducida (m)	Distancia interpretada (m)	Error absoluto (m)
0.26	0.346	0.086
0.28	0.321	0.041
0.3	0.314	0.014
0.32	0.319	0.001
0.34	0.3429	0.0029
0.4	0.4083	0.0083
0.5	0.5011	0.0011
0.6	0.6191	0.0191
0.7	0.759	0.059
0.8	0.8556	0.0556
0.9	0.936	0.036
1	1.185	0.185
1.1	1.3221	0.2221
1.2	1.6297	0.4297
1.3	1.89	0.59
1.35	2.608	1.258
1.4	2.61	1.21

Tabla 9.5: Detección de la pelota a través del eje X.

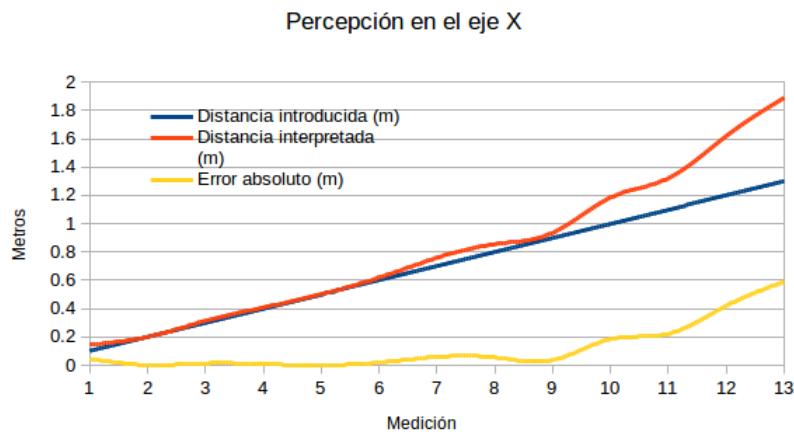


Figura 9.5: Gráfico de precisión del nodo ball\_pos para desplazamiento de la bola a través del eje X.

Distancia introducida (m)	Distancia interpretada (m)	Error absoluto (m)
-1.2	-1.46	0.26
-1	-0.99	0.01
-0.8	-0.75	0.05
-0.6	-0.53	0.07
-0.4	-0.33	0.07
-0.2	-0.16	0.04
0	0.003	0.003
0.2	0.159	0.041
0.4	0.3392	0.0608
0.6	0.544	0.056
0.8	0.749	0.051
1	1.021	0.021
1.2	1.458	0.258

Tabla 9.6: Detección a de la pelota a través del eje Y.

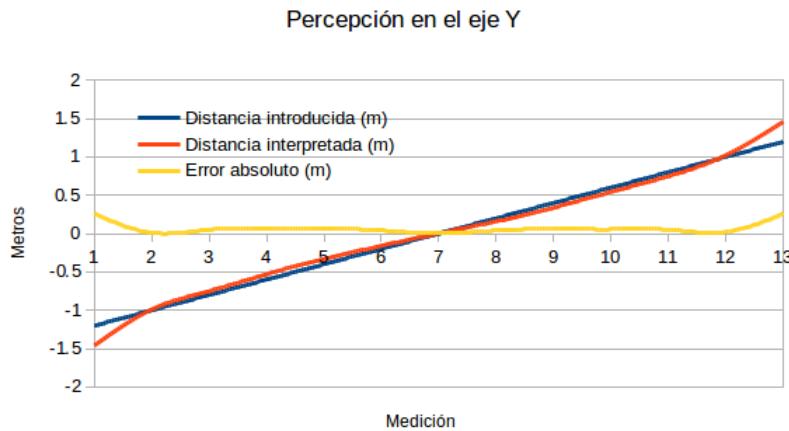


Figura 9.6: Gráfico de precisión del nodo ball\_pos para desplazamiento de la bola a través del eje Y.

### 9.2.3. Discusión

A la vista de los resultados, podemos concluir que las posiciones en las que el robot detecta con mayor precisión la pelota son entre 30 y 40 cm. con respecto al eje X, y entre -20 y 20 cm. con respecto al eje Y. Cuando las posiciones de la pelota se alejan de estos rangos, las estimaciones por parte del robot dejan de ser aceptables. De todos modos, cabe observar que dada la naturaleza de nuestro algoritmo, siempre y cuando se interpreten correctamente la orientación de la pelota (signo de la detección en el eje Y), el propio algoritmo se encarga de desplazar al robot linealmente de forma que al acercarse éste, las estimaciones se vuelven aceptables y el algoritmo de desplazamiento converge.

## 9.3. Detección de pelota y posicionamiento de robot

### 9.3.1. Objetivo de la prueba

Se pretende medir la efectividad del algoritmo de colocación del robot en la posición óptima de disparo. Para la realización de estas pruebas se colocará la pelota siempre en posiciones que se hayan demostrado como *detectables* en las pruebas anteriores (si la

Posicionamiento del Nao en diferentes situaciones de pelota (SÍ/NO)					
Eje X (m)	Eje Y (m)	Exp. 1	Exp. 2	Exp. 3	Exp. 4
0.4	0	SÍ	SÍ	SÍ	SÍ
0.5	-0.2	SÍ	SÍ	SÍ	SÍ
0.6	0.2	SÍ	SÍ	SÍ	SÍ
0.9	-0.3	SÍ	SÍ	SÍ	SÍ
1	0.3	SÍ	SÍ	SÍ	SÍ

Tabla 9.7: Detección a de la pelota a través del eje Y.

pelota no es detectada, obviamente el robot no podrá colocarse en base a su posición). Así, se modificará la posición relativa de la pelota con respecto al Nao, y se comprobará únicamente si el robot, mediante el algoritmo implementado, se acerca a ella y se coloca en posición de disparo. Cabe destacar también que se entiende por posición de disparo la posición óptima que se ha obtenido en la primera de las pruebas (de golpeo), esto es, (0.12, -0.04).

### 9.3.2. Resultados en simulación

Se han elegido ciertos puntos significativos, **donde la detección está garantizada por el experimento anterior**, y simplemente se ha anotado si el Nao se acerca correctamente y se posiciona para el golpeo. En la primera columna de la table 9.7 se puede ver la posición relativa en torno al eje X de la posición de colocación de la pelota, en la segunda lo mismo pero referido al eje Y, y en la tercera el éxito o fracaso de la prueba en esta posición.

### 9.3.3. Resultados con robot real

Se ha comprobado experimentalmente que el robot sigue los mismos patrones: si el balón es detectado, el algoritmo conduce al robot a éste. El problema que se ha detectado mediante las pruebas reales, radica en el posicionamiento último del Nao,

justo antes de golpear. Debido a que el rozamiento de los pies del robot con el suelo es mayor que el que se introduce idealmente en el simulador, el robot gira su orientación involuntariamente cuando se desplaza en el sentido del eje Y en las posiciones cercanas a la pelota.

Se han modificado, por segunda vez, para la prueba real, algunos de los parámetros del módulo. Se ha aumentado la cota de error permitido en el acercamiento, y se ha disminuido el desplazamiento efectuado por el robot para cada posición del balón recibida durante el acercamiento. Esto ha conseguido cierta optimización del algoritmo para el robot real, pero no ha conseguido solventar el problema del giro sobre sí mismo cuando el robot se encuentra en posiciones muy cercanas a la pelota.

#### 9.3.4. Discusión

Por segunda vez en las pruebas experimentales, se está obligado a retocar el diseño inicial, lo cual se encuadra en el contexto de la metodología ágil-iterativa seguida. Si en el golpeo se había visto la necesidad de modificar el tipo de patada para cada sentido del ángulo introducido en la misma, en esta ocasión, se estima oportuno el modificar las cotas de error y la cantidad de desplazamiento para contrarestar las inexactitudes del método propuesto cuando se lleva éste a un entorno real.



# Capítulo 10

## Conclusiones y trabajo futuro

En este capítulo se exponen las conclusiones del trabajo y también algunas de las mejoras futuras que se sugieren. Se comentan aquellas funcionalidades que se han alcanzado, así como los aspectos que por un motivo u otro no han podido ser finalmente incluidos en el diseño de este software.

### 10.1. Conclusiones

En base a los objetivos concretos propuestos en el capítulo 5, *Análisis*, podemos decir que el comportamiento que finalmente se ha diseñado para el robot humanoide Nao cumple con dichos objetivos en su mayoría, exceptuando la detección de un objeto externo que simbolizaría la portería, lo cual formaba parte de las funcionalidades que se habían predefinido.

Entre la lista de objetivos que sí se han cumplido podemos citar como puntos más importantes:

- **Se ha conseguido un golpeo estable**, que permite que el Nao realice la patada a la pelota sin comprometer su equilibrio, y dotando a esta patada de la fuerza necesaria para que el balón salga despedido a una distancia de varios metros con respecto al robot. Se hace notar también que *se ha conseguido que este golpeo sea variable* en el sentido que el robot, en base a un parámetro (ángulo medido sobre

el eje Y), es capaz de otorgar a la pelota una trayectoria previsible.

- **Se ha implementado una detección de balón efectiva.** El algoritmo de detección, unido al de búsqueda de la pelota, garantiza la detección de ésta, siempre y cuando se encuentre en el campo de visión de cualquiera de las dos cámaras.
- **Se ha propuesto un algoritmo de posicionamiento convergente.** El Nao es capaz de posicionarse con un margen de error notablemente reducido en el simulador, y funcional en las pruebas con el robot real.
- **Se ha diseñado un sistema distribuido,** ya que los nodos diseñados trabajan de forma independiente. Si uno de ellos genera un error, o se muere el proceso que lo sustenta, el resto de nodos no varían su comportamiento.
- **Se ha diseñado un sistema extendible.** La introducción de nuevos nodos que realicen otras funciones no debería interferir en el funcionamiento de los que ya se encuentran implementados.

## 10.2. Trabajo futuro

Como era de esperar, este trabajo tiene ciertas limitaciones y también se puede ampliar en varias direcciones. Entre todas ellas destacaremos las siguientes:

- **Detección de portería:** se entiende que sería la primera funcionalidad que actualmente no está presente en el software que debería implementarse, para un comportamiento completo de tiro de penaltis. Se sugiere que tal vez mediante la definición de un movimiento circular cuando el Nao está en las posiciones cercanas a la bola, con respecto a la portería detectada, el Nao sería capaz de apuntar con la patada a su objetivo aún partiendo de posiciones inicialmente aleatorias.
- **Incorporación de un algoritmo de búsqueda más extenso** del que actualmente dispone este software. En estos momentos el robot, ante la no-detección de la pelota en el entorno, mueve la cabeza en un movimiento de arriba a abajo,

y realiza un cambio de cámara. Se sugiere el diseño de un comportamiento que permita al Nao girar sobre su propio eje Z, e incluso caminar por el entorno en determinada trayectoria hasta encontrar una pelota.

- **Detección y evasión de posibles obstáculos** usando los detectores de ultrasonidos que ya están incorporados en el propio robot.
- **Incorporación de una cámara externa (Kinnect o similar)**, que permita al robot hacer triangulaciones más precisas sobre la posición de la pelota y la portería de lo que permiten las cámaras incorporadas en el Nao. Cabe destacar que esto además podría facilitar los algoritmos de alineación del robot con la pelota y la portería, ya que podríamos usar una imagen que englobase a los tres elementos en un mismo cuadro en todo momento, al contrario de lo que sucede con las cámaras del Nao.
- **Detección de otras formas y colores mediante OpenCV**. Como se ha comentado en el capítulo 3, *Valoración de hardware y software*, la detección de la pelota se hace mediante los módulos al efecto ya presentes en la API de Naoqi. Sin embargo, el uso de OpenCV, además de haberse constituido como un estándar *de facto*, permite mucha más libertad a la hora de efectuar la detección. Con el uso de OpenCV, podríamos hacer que el robot detectase otras formas y colores de pelota.
- **Incorporación de elementos de interacción con los humanos** del entorno, principalmente voz, gestos, sonidos y luces LED. También el uso de los sensores táctiles del Nao para modificar, cancelar o reactivar los distintos comportamientos o módulos implementados.
- **Desarrollo de una interfaz gráfica** para el software. Se propone el desarrollo de una interfaz gráfica, que además permita al usuario especificar parámetros de la aplicación (por ejemplo el ángulo de golpeo deseado) a través de controles amigables. En estos momentos el software debe lanzarse enteramente desde una terminal, como se expone en el Apéndice B, *Manual de usuario*.



## Apéndice A

# Protocolo de comunicaciones XML-RPC

### A.1. Introducción a XML-RPC

XML-RPC es un conjunto de herramientas que permiten la llamada a procesos a través del protocolo HTTP. Esto implica que los procesos pueden ser llamados desde máquinas remotas a través de Internet. Como su denominación indica, XML-RPC utiliza XML como formato de contención de mensajes, intentando con ello mantener la simplicidad del mecanismo, pero a su vez permitiendo el transporte en doble sentido de órdenes y respuestas complejas codificadas bajo *tags* XML.

Un mensaje XML-RPC es básicamente una petición de tipo HTTP-POST. El cuerpo del mensaje está codificado en XML, y las respuestas por parte del servidor están igualmente codificadas en XML. A continuación se adjunta un ejemplo de llamada a un procedimiento en remoto:

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181
<?xml version="1.0"?>
```

```
<methodCall>
  <methodName>ejemplo.suma</methodName>
  <params>
    <param>
      <value><i4>41</i4></value>
    </param>
    <param>
      <value><i4>54</i4></value>
    </param>
  </params>
</methodCall>
```

Y esta podría ser un posible respuesta por parte del servidor:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Thu, 29 May 2014 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>95</string></value>
    </param>
  </params>
</methodResponse>
```

## Apéndice B

# Manual de usuario

A continuación se definen una serie de pasos básicos que permiten la instalación y puesta en funcionamiento de este software. Se parte de la base de que el usuario tiene un sistema instalado compatible con el software sobre el que descansa la implementación, así, establecemos como requisitos de software:

- Sistema operativo Ubuntu 9.11 o posterior, o cualquier distribución GNU/Linux compatible con ROS.
- Intérprete Python 2.7.4.

Pasos a seguir en la puesta en funcionamiento del sistema:

1. Instalación de ROS a través de paquetes oficiales de Ubuntu, o compilándolo desde código fuente en el caso de otra distribución GNU/Linux. En la wiki oficial de ROS se pormenoriza este particular: <http://wiki.ros.org/ROS/Installation>
2. Instalación de la versión de prueba (o compra de la versión completa) de Webots, a través de la web del desarrollador: <http://www.cyberbotics.com/overview>
3. Compilación del código del proyecto a través de la herramienta de compilación de ROS: catkin. Se copia el contenido de la carpeta *src/* del proyecto a nuestro workspace ROS y se compila mediante el comando *catkin\_make* (ampliamente documentado en la wiki de ROS: [http://wiki.ros.org/catkin/commands/catkin\\_make](http://wiki.ros.org/catkin/commands/catkin_make))

4. Lanzamos Webots, que a su vez se encarga de lanzar Naoqi en el equipo local.  
Abrimos uno de los mundos incluidos en el software (por ejemplo, *nao-pfc.wbt*)

5. Configuramos la siguiente variable de entorno:

```
export LD_LIBRARY_PATH=/usr/local/webots/resources/projects/robots/nao/aldebaran/naoqi-  
runtime/lib:$LD_LIBRARY_PATH
```

6. Lanzamos el núcleo de ROS mediante el comando *roscore* (<http://wiki.ros.org/roscore>).
7. Damos permisos de ejecución a todos los nodos: *chmod +x catkin\_ws/src/nao-pfc/scripts/\*.py*
8. Lanzamos cada uno de los nodos por separado usando usando el intérprete python  
(por ejemplo, */\$ python ball\_pos.py*) o bien lanzamos varios con alguno de los  
ficheros .launch incluidos en la carpeta *nao-pfc/launch/* que automatizan esta  
tarea (por ejemplo, */\$ rosrun nao-pfc demo.launch*)
9. Deberíamos ver al robot respondiendo a la posición de la bola en el simulador.
10. Deberíamos ver el topic pos de ROS recibiendo mensajes desde el módulo de  
detección. Esto podemos verlo a tiempo real usando el comando *rostopic echo*  
*pos*.

## Apéndice C

# Contenido del DVD

En el DVD que se adjunta al proyecto, se puede observar la siguiente estructura:

- Carpeta **src**: contiene el código fuente comentado del proyecto. Este código basta para poder usar el software. Los pasos para la instalación y prueba se han expuesto en el apéndice B, *Manual de usuario*.
- **Memoria.pdf**, archivo en formato PDF (Portable Document Format) que contiene la presente memoria.
- Carpeta **video**: contiene algunos de los videos más significativos y que el autor encuentra más explicativos de cómo funciona el software implementado, tanto en simulación como en robot real.
- **webots\_7.4.3\_i386.deb**: archivo de instalación de la versión de prueba del simulador Webots, en su edición para sistemas basados en Debian y arquitecturas de 32bits.
- **webots\_7.4.3\_amd64.deb**: archivo de instalación de la versión de prueba del simulador Webots, en su edición para sistemas basados en Debian y arquitecturas de 64bits.
- **nao\_world.wbt**: archivo que contiene un mundo de Webots con un entorno similar a el dispuesto en el Robocup y que incorpora un robot Nao, una pelota y una portería.



## **Apéndice D**

### **Referencias bibliográficas**



# Bibliografía

- [1] URDF description <http://wiki.ros.org/urdf>
- [2] Rviz ROS documentation <http://wiki.ros.org/rviz>
- [3] Gazebo documentation <http://gazebosim.org/wiki/>
- [4] Webots documentation <http://www.cyberbotics.com/guide/>
- [5] Learning ROS for Robotics Programming Aaron Martinez, Enrique Fernandez  
Packt Publishing, 2013
- [6] ROS Wiki documentation <http://wiki.ros.org>
- [7] Naoqi API & documentation <https://community.aldebaran-robotics.com/doc/1-14/naoqi/>
- [8] OpenCV documentation <http://docs.opencv.org/>
- [9] Citius Wiki <https://wiki.citius.usc.es>
- [10] Diario Pravda, Sept. de 2006, <http://english.pravda.ru/science/mysteries/11-09-2006/84374-robots-0/>
- [11] Karel Capek, Rossums Universal Robots, Echo Library, 2010
- [12] SRI International's Artificial Intelligence Center, <http://www.ai.sri.com/shakey/>
- [13] Encyclopedia Astronautica, [http://en.wikipedia.org/wiki/Lunokhod\\_programme](http://en.wikipedia.org/wiki/Lunokhod_programme)
- [14] American Honda Motor Co. Inc., <http://www.ai.sri.com/shakey/>

- [15] Funcionamiento del robot humanoide Nao <http://www.aldebaran.com/en/humanoid-robot/nao-robot-working>
- [16] Funcionamiento de Naoqi y API: <https://community.aldebaran-robotics.com/doc/1-14/dev/naoqi/index.html>
- [17] Conceptos lógicos más importantes de ROS: <http://wiki.ros.org/ROS/Concepts>
- [18] Diseño del comportamiento de un portero de fútbol robótico [http://www.researchgate.net/publication/48989495\\_Diseo\\_del\\_comportamiento\\_de\\_un\\_portero\\_de\\_ftbo](http://www.researchgate.net/publication/48989495_Diseo_del_comportamiento_de_un_portero_de_ftbo)
- [19] Construcción y pruebas en Gazebo y V-Rep de un modelo para el robot humanoide Nao <http://tesi.cab.unipd.it/42989/>
- [20] Aprendizaje por Refuerzo Seguro para enseñar a un robot humanoide a caminar más rápido <http://e-archivo.uc3m.es/handle/10016/17996>
- [21] Robocup [http://www.robocup2014.org/?page\\_id=238](http://www.robocup2014.org/?page_id=238)
- [22] Berlin United <http://www.naoteamhumboldt.de/>