Implementación de técnicas de computación evolutiva a la programación automática de un robot

Pablo Gumiel Moreno
Yago Sáez Achaerandio
David Quintana Montero
Universidad Carlos III de Madrid

18 de junio de 2009

"¿Que si creo que las máquinas llegarán a pensar?
Téngalo por seguro. Yo soy una máquina,
usted es una máquina y ambos pensamos, ¿no cree?"
- Claude Elwood Shannon

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Motivación

Previo a la confección y desarrollo del presente proyecto fin de carrera, se ha llevado a cabo la fase que posiblemente sea la más importante de todas: la selección del mismo. Durante este período de elección entre las diferentes áreas y temáticas estudiadas durante los cursos que conforman el plan de estudios, dos preguntas han sido las que han facilitado esta selección. Dos sencillas preguntas que han servido para orientar todo el trabajo que se plasma en el presente documento y que han tenido mucha relevancia durante los años de estudio de la carrera, en lo que a selección de asignaturas se refiere.

La primera de estas preguntas, ¿qué es la robótica? Si nos atenemos a la definición formal de la Real Academia de la Lengua, la robótica es "*la técnica que aplica la informática al diseño y empleo de aparatos que, en sustitución de personas, realizan operaciones o trabajos, por lo general en instalaciones industriales*". Realmente, esta definición no es del todo precisa, ya que la robótica es multidisciplinar, puesto que combina diversas áreas como son: la mecánica, la electrónica, la informática, la inteligencia artificial, la ingeniería de control, el álgebra y la física. Llegados a este punto, además, se puede afirmar que la robótica es también una ciencia. La ciencia encaminada a diseñar y construir aparatos y sistemas capaces de realizar tareas propias de un ser biológico.

En contra de lo que pueda pensarse, la robótica es muy antigua, incluso más antigua que cuando el escritor checo Karel Čapek acuñó el termino *robot* en su obra "*Rossum's Universal Robots / R.U.R.*", en el año 1921. Unos de los primeros autómatas que se pueden considerar parte de la robótica fueron unas estatuas realizadas por Amenhotep, rey de Etiopía, antes del año 1500 a.C. Dichas estatuas emitían ruidos al ser iluminadas por la luz del sol. En 1260, aparece el primer robot humanoide programable,el cual era capaz de servir bebidas, diseñado por el árabe Al-Jazari. Hay que avanzar hasta el año 1946 para encontrar comportamientos biológicos simples en estos robots, cuando William Grey Walter, exhibió a los robots Elsie y Elmer, que eran capaces de localizar el lugar donde debían cargarse cuando se encontraban con la batería baja. Posteriormente, en 1961 se implanta el primer robot industrial. Durante los últimos 50

años se han producido grandes avances, llegando hasta nuestros días con robots como ASIMO y AIBO, reflejo del avance de esta ciencia.

La segunda pregunta que motivó la selección del presente proyecto fin de carrera fue: ¿qué es la inteligencia artificial? Se denomina inteligencia artificial a la rama de la informática dedicada al desarrollo de agentes racionales no biológicos. Entendiendo por agente como cualquier *cosa* capaz de percibir su entorno (recibir entradas), procesar tales percepciones y actuar en su entorno (proporcionar salidas). Por lo tanto, y de manera más específica, la inteligencia artificial es la disciplina que se encarga de construir procesos que al ser ejecutados sobre una arquitectura física producen acciones o resultados que maximizan una medida de rendimiento determinada, basándose en la secuencia de entradas percibidas y en el conocimiento almacenado en tal arquitectura. Dentro de la inteligencia artificial existen diferentes técnicas, como pueden ser las redes de neuronas, la lógica formal y los algorítmos genéticos.

¿Qué pueden aportar cada una de estas dos respuestas? Se ha demostrado que cada una de estas dos tecnologías, robótica e inteligencia artificial, han constituido un gran avance de manera individual para la sociedad. Pero sin duda, los mayores avances y mejoras vienen de la fusión de ambas tecnologías. Cada vez, se busca más autonomía en los robots y que sean capaces de adaptarse a un entorno cambiante. Esta capacidad de adaptación se puede resumir como una búsqueda y una optimización del comportamiento del robot para obtener una solución eficiente al problema planteado. Por lo tanto, esta búsqueda y optimización se puede conseguir a través de la aplicación de técnicas de inteligencia artificial a la robótica. Esto es conocido como robótica evolutiva, es decir, un área de la robótica autónoma en la que se desarrollan los controladores de los robots mediante la evolución, al usar algoritmos genéticos. Este área de investigación es muy interesante en ámbitos en los que no sea posible la presencia humana, como pueden ser: conflictos bélicos, la exploración espacial, la exploración submarina, el rescate en catástrofes... También es de gran interés el desarrollo de herramientas que faciliten tareas, como son: el diseño de prótesis, reconocedores de patrones, sistemas de autoguiado...

Todo lo expuesto anteriormente ha motivado la elección del presente proyecto fin de carrera, con el objetivo principal de realizar un controlador para un robot de manera automática, por medio de técnicas de inteligencia artificial.

Capítulo 2

Estado del arte

A lo largo del siguiente capítulo se detallarán diferentes artículos y publicaciones sobre la aplicación de técnicas de inteligencia artificial al control y guiado de diferentes tipos de robots. Se hará un estudio de estas técnicas y del campo sobre el que trabajan para determinar las líneas de investigación del presente proyecto de fin de carrera.

2.1. Genetic Programming Approach to the Construction of a Neural Network for Control of a Walking Robot [1]

Este artículo es el primer estudio publicado sobre la aplicación de técnicas de programación genética al desarrollo del movimiento en robots. Ha sido citado en numerosos artículos y es reconocido como el primero que versa sobre este tema. El trabajo realizado por los investigadores de la University of Southern California se basa en la evolución de un motor complejo de generación de patrones (MPG), formado por una red de neuronas cuyos pesos se determinan mediante un algoritmo genético.

El objetivo del trabajo es la evolución de una red de neuronas que genere la secuencia de impulsos adecuada para dirijir las patas de un robot hexápodo con 12 grados de libertad de movimiento. Para el diseño del controlador del robot se utilizarán técnicas de optimización por medio de un algoritmo genético.

Para el desarrollo del estudio se realizó una pre-evolución del controlador con una función de *fitness*, lo cual llevó al sistema a trabajar en un espacio de búsqueda más óptimo y a tener unas condiciones más fáciles de satisfacer. Una vez que el sistema se encuentra en el entorno apropiado para encontrar una solución se utiliza una función de *fitness* más compleja para refinarla.

Los experimentos se realizaron con un robot hexápodo llamado *Rodney*, el cual tiene seis patas con dos grados de libertad cada una de ellas. Los motores del robot aportan funciones de giro y de elevación a cada una de las patas. Para el simulador del algoritmo genético se utilizó el motor *GENESIS*, y la función de evaluación transforma el código genético en la descripción de la red de neuronas. El resultado se utiliza de nuevo como *feedback* para el algoritmo genético, como se puede observar en la Figura 2.1.

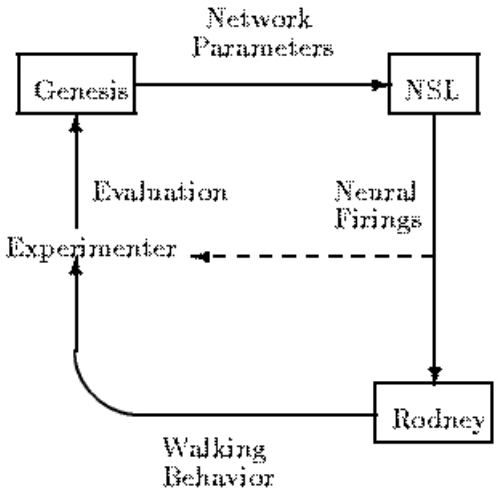

Figura 2.1: Configuración del experimento con Rodney

El primer paso del aprendizaje genético se basa en descubrir un conjunto de parámetros que implementen de una manera razonable un oscilador de la pata del robot. Simplemente, con conectar los osciladores individuales del robot, no se produjo un andar efectivo en *Rodney*, debido a que no había ninguna manera de coordinarlos. El código genético especifica cuatro parámetros que forman parte del circuito: los dos pesos y los dos umbrales. La fase de evaluación está dividida en un conjunto de etapas evolutivas con la intención de dirigir la red de neuronas a un punto donde la población se encuentre cercana a unos valores determinados. Una vez que la población comienza a acercarse a un *fitness* determinado se pasa a la siguiente fase.

El segundo paso no sólo se centra en la circuitería sino también en las conexiones entre los osciladores de las patas del robot. Se usan cuatro nuevos parámetros para representar los conectores. En este punto de la evaluación, el caminar hacia atrás también es recompensado en el *fitness*, de la misma manera que al caminar hacia delante, ya que ambos necesitan parámetros muy similares y se puede realizar una transición muy simple de uno a otro.

Resultados

El genoma utilizado para el experimento fue de 65 bits: 8 bits para cada uno de los parámetros de los osciladores y de los conectores, tomando valores de -8 a 8 codificados en *código gray*, el último bit determina el mapeado del giro de la pata, lo cual hará que en un caso se mueva hacia

delante y en el otro hacia atrás. El tamaño de la población será de 10 individuos, la probabilidad de mutación del 0.4, la probabilidad de cruce del 0.1, y elitismo de 2.

Con estos parámetros para el algoritmo genético en la primera fase se necesitan de 7 a 17 generaciones antes de que la mitad de la población empiece a oscilar cerca del límite marcado. En todos los casos los parámetros producen una solución en la que las dos neuronas realizan un giro de 90 o de 270 grados, con igual probabilidad para ambos casos. En la segunda fase se necesitan entre 10 y 35 generaciones para que se alcance el máximo, observándose que la media de todos los individuos es significativamente menor que el mejor individuo de cada generación, debido a la sensibilidad del algoritmo a la alta mutación. Las gráficas de ambas fases se pueden observar en la Figura 2.2.

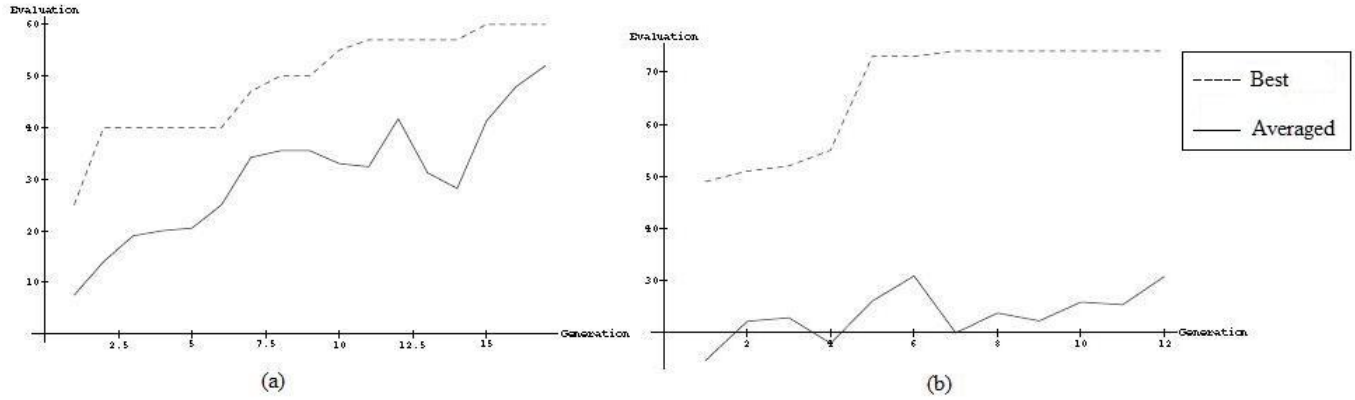

Figura 2.2: Comparación fitness medio - mejor fitness: primera etapa (a) y segunda etapa (b)

Finalmente, se consigue que *Rodney* aprenda a caminar de manera *trípode*, es decir, dejando siempre apoyadas en el suelo tres de sus seis patas moviéndose a la vez la pata delantera izquierda, trasera izquierda y media derecha. Esta manera de caminar es muy frecuente en los insectos. Si se elimina la penalización por girarse de la función de *fitness* y simplemente se basa en la distancia recorrida, el robot tiende a realizar un giro entre 90 y 120 grados. Pero el resultado más sorprendente, según los autores, es el hecho de que cambiando un parámetro de la función de *fitness* el robot camina siempre hacia atrás en vez de hacia adelante, posiblemente debido al ángulo de empuje de las patas contra el suelo.

Dentro de las conclusiones destaca la afirmación de que los algoritmos genéticos son difícilmente aplicables debido al gran espacio de búsqueda y el elevado coste de evaluación para la época (año 1992). El uso de la evolución por etapas ayuda a la convergencia de la solución. Por último, concluyen que los algoritmos genéticos tienen la ventaja de que no es necesario el conocimiento de la dinámica del robot para poder desarrollar un controlador efectivo y que si estas técnicas son desarrolladas lo suficiente, podrán liderar la metodología para construir sistemas robóticos complejos (lo cual se podrá ver en artículos realizados años más tarde y que se plasman en este proyecto fin de carrera).

Como futuros estudios se plantea explorar una red de neuronas más compleja para conseguir un diseño más complejo, incluyendo extensiones como: el uso de entradas de control para la red de neuronas, diferentes funciones de *fitness*, evolución de sistemas de percepción y extrapolación de los resultados a robots cuadrúpedos.

2.2. Autonomous Evolution of Gaits with the Sony Quadruped Robot [2]

Los autores de este artículo pertenecen a un equipo de desarrollo de Sony, empresa creadora y distribuidora del robot Aibo. En él tratan sobre la implementación de un algoritmo evolutivo autónomo para el desarrollo de métodos de locomoción para dicho robot, de manera que puedan ser evaluados por medio de los sensores propios de Aibo.

Antes de diseñar las técnicas evolutivas, el equipo de desarrollo programó manualmente la manera de caminar del robot. Se creó una manera de andar lenta, aproximadamente 5 metros por minuto, y una manera de andar entre la anterior y el trote, que recorría aproximadamente 6 metros por minuto.

La principal diferencia entre este estudio y el realizado en [1] estriba en que la evolución en el presente artículo es completamente autónoma y se desarrollan maneras de andar dinámicas, ya que en [1] el cálculo del *fitness* se hacía de manera no automatizada, necesitando la presencia humana, y en varias fases. El desarrollo autónomo conlleva coordinar diferentes sensores del robot y esto ya de por sí es un gran problema. Como se verá posteriormente en los resultados, los controladores evolutivos fueron mucho mejores que los desarrollados manualmente.

El robot Aibo presenta tres grados de libertad en cada pata, tres en la cabeza y uno en el rabo, sumando en total 16 grados de libertad. El movimiento de las patas está controlado por un módulo de locomoción, el cual es capaz de controlar al robot por medio de un conjunto de parámetros. Este módulo reevalua los motores cada 20 milisegundos para calcular la siguiente fase del movimiento y recupera al robot de eventuales caídas. En la Figura 2.3, se muestran esta serie de parámetros, que a su vez serán los genes de los individuos utilizados en el algoritmo evolutivo, el cual será lanzado en el robot.

La población inicial se crea con una distribución uniforme dentro del rango de búsqueda dado en la Figura 2.3. Los valores obtenidos para el conjunto de parámetros en los individuos de la población inicial pueden hacer que se creen individuos que en vez de andar se caigan. Estos individuos son suplidos por otros generados aleatoriamente, de forma que se tenga una población inicial en la que no se caiga ningún individuo. Posteriormente, se realiza una selección por el método de torneo. Primero se decide si se cruzarán los individuos o se mutarán. Para el caso del cruce se seleccionan aleatoriamente 3 individuos, de los cuales los padres serán los de mayor *fitness*, y el de menor será reemplazado por el individuo resultante del cruce de los padres. En el

parameter	unit	initial range
body center x	mm.	85 - 95
body center z	mm.	-5 - 5
body pitch	degrees	-5 - 5
all legs y	mm.	5 - 25
front legs z	mm.	24 - 40
rear legs z	mm.	15 - 29
step length	n.a.	80 - 220
swing height	mm.	15 - 29
swing time	ms.	200 - 400
swing mult.	n.a.	1.5 - 2.5
switch time	ms.	500 - 900
ampl body x	mm.	-2 - 2
ampl body y	mm.	0 - 20
ampl body z	mm.	-2 - 2
ampl yaw	degrees	-2 - 2
ampl pitch	degrees	-3 - 3
ampl roll	degrees	-3 - 3
min. gain	n.a.	25 - 175
shift	degrees	60 - 120
length	degrees	90 - 150

Figura 2.3: Conjunto de parámetros del módulo de locomoción de Aibo

caso de la mutación, se seleccionan 2 individuos aleatoriamente. Ambos casos tienen la misma probabilidad de suceder y se utiliza una distribución gausiana para decidir que genes del 1 al 8 se mutarán. Posteriormente se muta ese gen añadiéndole un número aleatorio en un rango entre -1 y 1. Una vez finalizados los cruces y las mutaciones se obtiene una nueva población para la siguiente generación.

Para la evaluación de cada individuo el robot mide su distancia a la pared y se le hace caminar durante una cantidad de tiempo determinada. Posteriormente, se calculan dos valores para usar en la función de *fitness*: la distancia a la pared y el ángulo de giro respecto a la posición inicial. Si el individuo se cae, la puntuación obtenida es 0. Además, si el individuo es incapaz de encontrar la linea de color, es decir, que el ángulo de giro es grande, también se le da una puntuación de 0. La puntuación final de cada individuo es la media de las puntuaciones obtenidas en tres ejecuciones, tras usar los valores de la distancia y el ángulo dentro de la función de *fitness*.

Resultados

Los resultados se diferencian en los obtenidos para el trote y los obtenidos para el paso. En el caso del trote se usó una población de 30 individuos y se hicieron 21 generaciones. En la generación inicial muchos individuos tendían a caminar hacia atrás o en curva, el mejor individuo andaba 26 centímetros. Al final de la evolución, en la mayor parte de la población, la manera de caminar era más recta y refinada. El mejor individuo andaba 6.5 metros por minuto. En el caso del paso se usó una población de 30 individuos, pero fueron necesarias 84 generaciones aleatorias para conseguir una población inicial en la que los individuos no se cayeran. Al igual que en el trote, la

mayoría de los individuos de la población inicial tenían un *fitness* bajo. Se utilizó un parámetro de edad para evitar super-individuos. Este parámetro se reinicia cada 4 generaciones. Tras 11 generaciones se obtiene el mejor individuo que es capaz de desplazarse a 10.2 metros por minuto. Los resultados de ambos experimentos se pueden observar en la Figura 2.4. En ella se plasma la evolución del *fitness* medio de la población y el del mejor individuo en cada generación, tanto para el caso del trote como para el del paso.

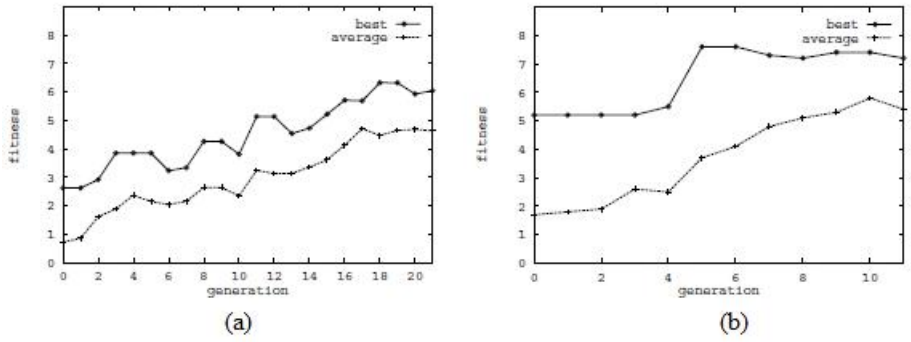

Figura 2.4: Variación del fitness para el trote (a) y para el paso (b)

Como conclusiones de este experimento se puede observar que el mejor individuo del paso es más rápido que el mejor individuo del trote, lo cual puede ser debido al diseño del hardware del robot. En el trote, el movimiento del robot es más inestable, puesto que se deja levantada una pata de cada lado. Debido a lo expuesto anteriormente, los autores llegan a la conclusión de que el trote dinámico a dos patas no es estable y eventualmente puede caer debido a que es necesaria una tercera pata para estabilizar al robot. Al moverse rápido, las patas que hay apoyadas en el suelo se van intercalando y el robot no tiene tiempo de apoyar una tercera. Durante la evolución, los primeros individuos se movían muy lentamente y apoyándose en una tercera pata. Se sufrieron muchas averías en el robot durante los 4 meses que duró el experimento. Por lo tanto, concluyen que utilizar un simulador es la mejor opción. El algoritmo evolutivo ha generado muy buenos resultados, tanto en el trote como en el paso, superando a los generados manualmente, tal y como se observa en la Figura 2.5.

	initial pop best fitness	overall best fitness	overall best speed
pace	5.2	7.6	10.2 m/min
trot	2.6	6.3	6.5 m/min

Figura 2.5: Comparativa entre el resultado del algoritmo genético y el del manual

2.3. Evolving Robust Gaits with AIBO [3]

Este artículo esta basado en los experimentos realizados con el robot de Sony Aibo, en su versión ERS-110, la primera versión comercial del robot. Con anterioridad a este artículo se realizó un algoritmo evolutivo para la adquisición automática del conjunto de parámetros para que el robot pudiera andar [2]. El problema del citado estudio es que las pruebas se realizan en un tipo de escenario determinado y que los resultados obtenidos pueden variar si cambia el entorno. Además, con el cambio de versión del robot, cambia el hardware del mismo y, se debe cambiar el algoritmo para que el robot ande, lo cual puede llevar muchas horas para hacerlo manualmente. Para simular diferentes escenarios se realizaron las pruebas en una alfombra similar a la de [2] y en la misma alfombra con trozos de plástico simulando obstáculos.

El robot Aibo, en esta versión, presenta tres grados de libertad en cada pata, tres en la cabeza, dos en el rabo y un actuador en cada oreja, en total 19 grados de libertad. El movimiento de las patas está controlado por un módulo de locomoción, el cual es capaz de controlar al robot por medio de un conjunto de parámetros. En esta versión son 61 los parámetros necesarios, aunque serán reducidos a 20, puesto que algunos parámetros toman valores fijos. En la Figura 2.6, se muestran esta serie de parámetros, que a su vez serán los genes de los individuos utilizados en el algoritmo evolutivo. La principal diferencia con los parámetros de [2] es la relación de variación entre el movimiento de las diferentes patas: L-R para la diferencia entre izquierdas y derechas y F-H para la diferencia entre delanteras y traseras. De esta manera, la pata delantera derecha empieza siempre en 0.0, la delantera izquierda comienza en L-R, la trasera derecha en F-H y la trasera izquierda en L-R + F-H.

El método de evaluación de cada individuo será el mismo que el utilizado en [2], es decir, se tendrá en cuenta la distancia a la pared y el ángulo de desviación con respecto al ángulo de inicio. Ambos valores serán tenidos en cuenta en la función de *fitness*.

Resultados

Como se comentó anteriormente, los experimentos fueron realizados en dos superficies diferentes: en una alfombra y en una alfombra con tiras de plástico. Se ejecutó tres veces el algoritmo evolutivo en cada una de ellas. Además, el mejor individuo evolucionado en cada superficie fue probado en diferentes superficies para comprobar cual obtenía mejores resultados. En cada una de las superficies se ejecutó el algoritmo genético durante 500 generaciones, lo cual se demoró más de 25 horas. Los resultados de ambos experimentos se pueden observar en la Figura 2.7. En ella se observa la evolución del *fitness* medio de la población y el del mejor individuo en cada generación, tanto para el caso de la superficie lisa como para el de la superficie rugosa.

Parameter List For A Gait

parameter	unit	initial range
body center x	mm.	105 - 125
body center z	mm.	-10 - 10
body pitch	degrees	-10 - 10
posture center x	mm.	0 - 20
all legs y	mm.	-5 - 15
front legs z	mm.	10 - 30
rear legs z	mm.	-5 - 15
step length	n.a.	60 - 100
swing height front	mm.	25 - 45
swing height rear	mm.	25 - 45
swing time	ms.	460 - 540
swing mult.	n.a.	3 - 5
ampl body x	mm.	-10 - 10
ampl body y	mm.	-25 - -5
ampl body z	mm.	-20 - 0
ampl yaw	degrees	-10 - 10
ampl pitch	degrees	-10 - 10
ampl roll	degrees	-5 - 15
L-R	n.a.	0.25 - 0.5
F-H	n.a.	0.5 - 0.75

Swing Starting Times for Different Gaits

Leg	Crawl	Trot	Pace	Skip
right foreleg	0.0	0.0	0.0	0.0
left foreleg	0.5	0.5	0.5	0.0
right hind-leg	0.75	0.5	0.0	0.5
left hind-leg	0.25	0.0	0.5	0.5

Offset Values for Different Gaits

Parameter	Crawl	Trot	Pace	Skip
L-R	0.5	0.5	0.5	0.0
F-H	0.75	0.5	0.0	0.5

Figura 2.6: Conjunto de parámetros del módulo de locomoción de Aibo ERS-110

2.4. Autoguiado de robots móviles mediante redes de neuronas [4]

El objetivo del trabajo realizado es la implementación de una estrategia de autoguiado de un robot móvil en entornos desconocidos empleando como núcleo de decisión una red neuronal. Se persigue que el robot sea capaz de moverse por un entorno de manera que no colisione con ningún objeto mientras encuentra una meta dada, intentando aprovechar la capacidad de generalización de las redes neuronales para solventar los posibles problemas que se encuentran en la navegación de un robot. El estudio se ha realizado en un entorno de Matlab y posteriormente sobre el microbot PICBOT3 de la empresa *Microsystems Ingeeniering*, Figura 2.8.

Durante la generación del sistema de guiado se pretende que el robot se mueva como una persona, captando información de su alrededor por medio de sensores y según ello, decida la acción que realizará. Para conseguirlo se utiliza la capacidad de generalización que ofrecen las redes de neuronas. El entorno de navegación será un cuadrado de 5,40 metros de lado, representado po una imagen de 120x120 píxeles que contiene una cantidad de obstáculos aleatorios que deberá esquivar el robot. El robot sólo puede realizar tres movimientos: avance sin giro, giro de 45º y avance, giro de -45º y avance.

La red neuronal es un perceptrón multicapa (*Multi-Layer Perceptron MLP*), la cual es adecuada

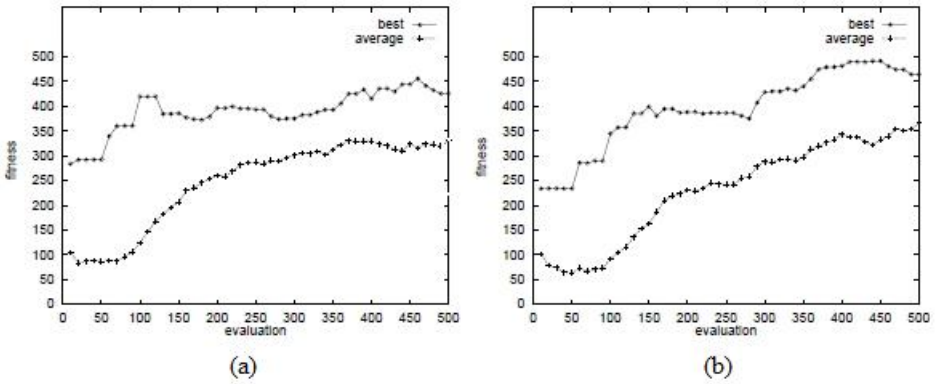

Figura 2.7: Variación del fitness para la superficie lisa (a) y para la rugosa (b)

Figura 2.8: Microbot PICBOT3

para los problemas de clasificación. Esta red está formada por seis entradas, cuatro neuronas de capa oculta y tres de salida, con una función de activación de tipo sigmoidal. Debido a que el microcontrolador que se implementó tiene unas características limitadas para realizar las operaciones, se entrenó también una red de perceptrón simple. Ambas redes se pueden ver en la Figura 2.9.

Para el entrenamiento de las redes neuronales se utilizaron unas series de trayectorias en el entorno del robot de las cuales se extraen todos los parámetros de los vectores de entrada y de salida de la red. Estas trayectorias, son rastros dibujados en la imagen del entorno, como ejemplos de las posibles trayectorias que podría realizar el robot. Además, hay que destacar que estas trayectorias son continuas y que no pasan dos veces por el mismo punto. Para extraer los datos de las trayectorias se programaron funciones de Matlab específicas y para la creación, entrenamiento y test de la red, la herramienta *Neural Networks* de Matlab, con la cual se puede implementar tanto el MLP como el perceptrón simple.

En la implementación en el robot se cambiaron dos aspectos con respecto a la versión de la

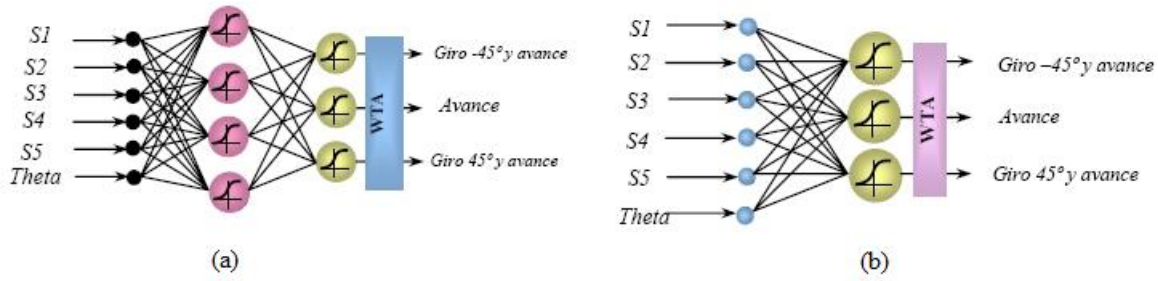

Figura 2.9: Estructura del MLP (a) y estructura del perceptrón simple (b)

simulación: ya que el microcontrolador es de 8 bits sólo se pueden usar números enteros entre -127 y 128, por lo tanto se tuvo que normalizar los pesos de la red de neuronas. El segundo cambio fue la aproximación de la función sigmoidal a una función lineal a tramos.

Resultados

Las primeras trayectorias y resultados se simularon en un entorno sin obstáculos, en el cual el robot se orientaba correctamente hacia la meta, realizando varias pruebas cambiando origen y meta. Se obtuvieron resultados aceptables en cuanto a la optimización de la trayectoria tanto con el MLP como con el perceptrón simple.

Posteriormente se añadieron obstáculos y se simularon nuevas trayectorias, como se ve en la Figura 2.10. Aunque las soluciones propuestas son diferentes se cumple el objetivo fijado.

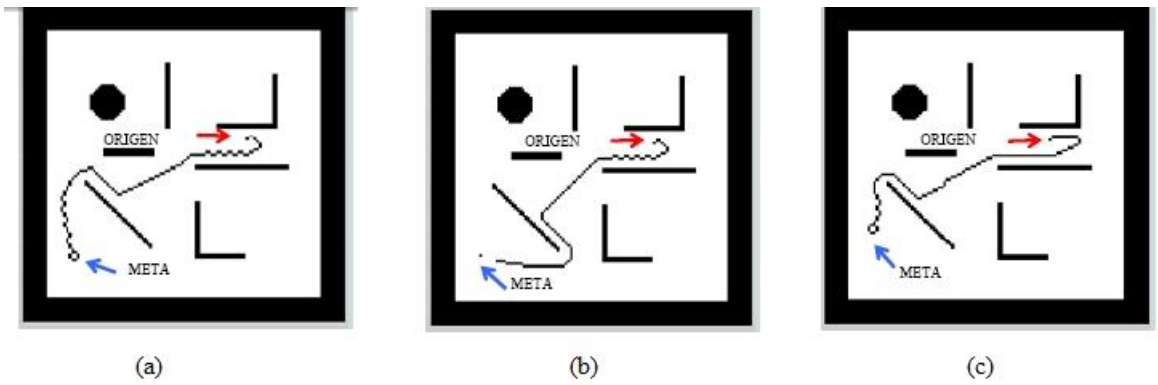

Figura 2.10: Trayectorias: MLP (a), perceptrón simple (b) y aproximación (c)

Como principales conclusiones del estudio los autores comentan los problemas que hubo en el ajuste de los ciclos de entrenamiento de las redes y en la selección del número de neuronas de la capa oculta. Se observó que ambas redes ofrecían soluciones diferentes, aunque aceptables en los dos casos. Se verificó que la red era capaz de generalizar, permitiendo simular trayectorias con distintos orígenes y metas y con entornos variados. Finalmente, se realizaron simplificaciones para poder aplicar el sistema a PICBOT3, comprobándose que funcionaban correctamente en

un entorno real (fuera del simulador).

Como futuras líneas de trabajo se han fijado: ampliar las acciones de control que realiza el robot, considerar que los sensores no sólo detectan la presencia o ausencia de un objeto, sino que miden también la distancia y añadir sensores a la parte trasera del robot para poder navegar marcha atrás.

2.5. Generando un agente robótico autónomo a partir de la evolución de sub-agentes simples cooperativos [5]

El objetivo del estudio al que hace referencia el artículo es la generación de un agente robótico autónomo a partir de la unión de sub-agentes simples implementados por redes de neuronas sencillas. Estos sub-agentes deben cooperar por medio de neuro-evolución. La cooperación entre ellos no será especificada de antemano, sino que serán los propios sub-agentes los que decidan como realizarla.

Existen diversos estudios en los que se realiza una división en sub-agentes especializados en tareas, dando lugar a los sistemas multi-agente, más generalmente conocidos como *MAS (Multi-Agent System)*, los cuales suelen estar programados de antemano. La propuesta realizada por los autores, R. A. Tellez y C. Angulo, se basa en la división en sub-agentes formados por controladores muy simples y no pre-programados. Cada uno de estos sub-agentes estarán a cargo de un sensor o de un actuador del robot. Como punto importante, no se codifica el objetivo de los mismos, tan solo se les facilita los medios necesarios para que puedan realizar la tarea que se les ha asignado.

El objetivo de los experimentos realizados en el estudio presentado por este artículo es demostrar que un controlador neuronal distribuido basado en sub-agentes simples es posible y, además, se comporta de manera más inteligente que un controlador neuronal centralizado, lo cual puede ser aplicado al control de movimiento de manos robóticas o brazos mecánicos. Todos los experimentos realizados se han basado en un entorno simulado dado que el robot físico no estaba construido en el momento de realizar el estudio.

El robot simulado consiste en dos ruedas motrices, que serán los actuadores, y en dos sensores de infrarojos. Cada sub-agente estará formado por el hardware de un sensor o de un actuador y por un microcontrolador que implementa una red neuronal artificial de capacidad reducida.

La red de neuronas utilizada será una red retroalimentada compuesta por una capa oculta con cuatro neuronas de entrada y una de salida, Figura 2.11. Las neuronas de entrada contienen las salidas de las redes neuronales de los otros sub-agentes y la neurona de salida proporciona el valor final procesado, que es enviado también al resto de sub-agentes. El resultado de usar este modelo es la colaboración de elementos neuronales muy simples y que aprenden en conjunto.

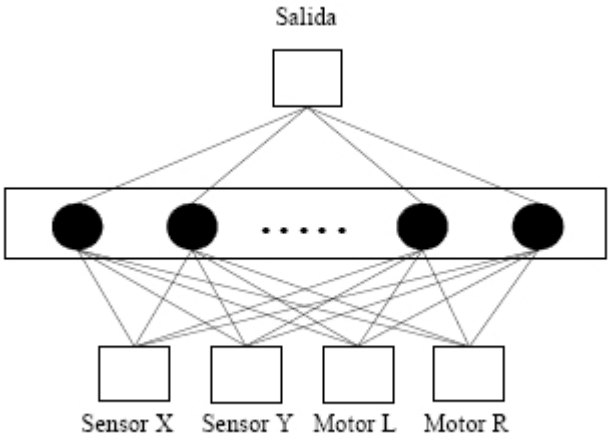

Figura 2.11: Red de neuronas con capa oculta de cuatro entradas y una salida

El algoritmo de aprendizaje utilizado es un método de neuro-evolución no supervisado, denominado *Enforced SubPopulations (ESP)*. Este método esta diseñado para evolucionar al mismo tiempo diversas redes neuronales con un objetivo común.

Resultados

Para todos los experimentos realizados en este estudio se ha utilizado un robot simulado bajo el entorno matemático Scilab, por lo tanto, puede que no se correspondan en su totalidad al resultado que se obtenga con el robot real. Por lo tanto, los resultados obtenidos están condicionados a este hecho.

Para el sistema multi-agente se evaluaron todas y cada una de las redes neuronales de los sub-agentes con el objetivo de encontrar un objeto rectangular situado en su espacio y que orbite alrededor del mismo. La función de evaluación o *fitness* premia al robot que se mueve, detecta un objeto a su izquierda y a la vez no detecta nada enfrente. El resultado fue que las redes neuronales consiguieron controlar al robot de la manera deseada.

Además se realizó la comparación con un controlador neuronal central para el robot, es decir, que estaría controlado por una red neuronal manejando todos y cada uno de los sensores y actuadores del agente robótico. Comparando las generaciones con el método anterior, los resultados son que el robot controlado por sub-agentes aprendía el proceso en aproximadamente la mitad de generaciones y, además, obteniendo siempre un *fitness* superior al mono-agente.

Por último, se utilizó una red neuronal para el control de los sensores, aún teniendo en cuenta que el coste computacional en comparación con la mejora obtenida pudiera ser muy alto. Para ello, se realizaron dos nuevos experimentos. En uno de ellos se evolucionó un controlador multi-agente en el cual no existen los sub-agentes de los sensores. El resultado fue que lograba el objetivo marcado pero el *fitness* máximo obtenido era menor que en el caso de cuatro sub-agentes. En el

segundo experimento se simuló el mal funcionamiento de uno de los sensores. Incluso con este defecto era capaz de lograr los objetivos marcados. Comparando ambas funciones de aprendizaje para un mismo sub-agente se ve que depende del comportamiento de su sensor asociado y de la situación en la que se encuentre, Figura 2.12.

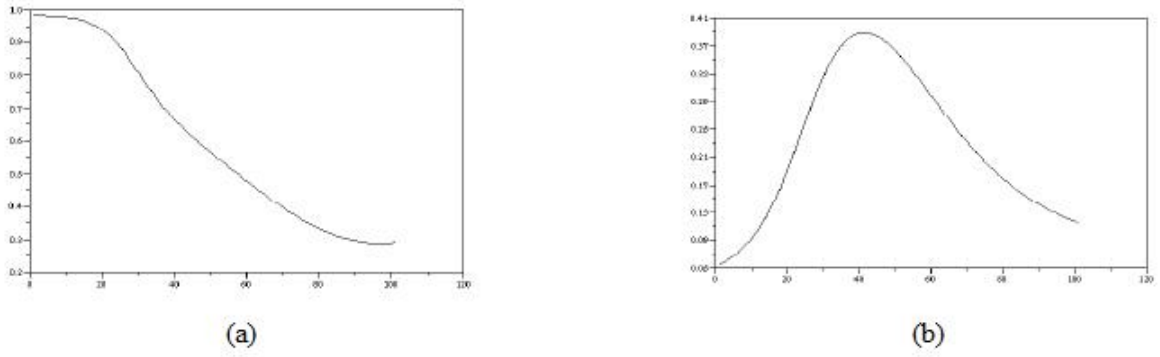

Figura 2.12: Funciones aprendidas por el sensor correcto (a) y el sensor defectuoso (b)

Por tanto, este artículo concluye que es posible obtener un agente inteligente a partir de la unión de sub-agentes simples sin necesidad de especificar qué es lo que debe hacer cada uno de ellos, aprendiendo más rápido y mejor que el que esta controlado por una única red neuronal compleja. También concluye, que es beneficioso utilizar sub-agentes en los sensores a pesar del coste computacional que ello conlleva. La justificación que ofrecen se basa en la tolerancia a fallos y en la escalabilidad.

2.6. Autonomous Evolution of Dynamic Gaits With Two Quadruped Robots [6]

La publicación de este artículo se realizó el 3 de junio de 2005 en el *"IEEE Transactions on Robotics, Vol. 21"*. En él se recogen los resultados obtenidos en dos estudios anteriores: "*Autonomous Evolution of Gaits with the Sony Quadruped Robot"[2]* y "*Evolving Robust Gaits with AIBO"[3].* A lo largo del artículo se explican las técnicas evolutivas utilizadas y los resultados obtenidos para dos modelos diferentes de Aibo: el prototipo OPEN-R y la versión ERS-110.

De la introducción del artículo cabe destacar que se utilizan algoritmos evolutivos para la optimización de un vector de parámetros que posteriormente será transmitido al sistema de movimiento del robot. Todo el proceso evolutivo es controlado por el procesador del robot y cada vector de parámetros es evaluado usando los sensores del propio robot. Además, no sólo se destaca que se ha conseguido realizar el movimiento del robot con algoritmos evolutivos, sino que en ambos casos (para los dos modelos) se ha evolucionado un sistema de caminar que es

mejor que los desarrollados manualmente y que el realizado para la versión ERS-110 ha sido incluido en la versión comercial de Aibo.

Una de las diferencias entre ambas versiones del robot estriba en los grados de libertad, teniendo 16 el prototipo OPEN-R (tres en cada pata, tres en la cabeza y uno en el rabo) y 19 la versión ERS-110 (los del prototipo OPEN-R más uno adicional en el rabo y un actuador en cada oreja). El movimiento de las patas está controlado por un módulo de locomoción, que funciona como se explicó en [2] y [3] con los parámetros que se pueden ver en la Figura 2.3 para el prototipo OPEN-R y en la Figura 2.6 para la versión ERS-110. Estos conjuntos de parámetros serán los genes de los individuos utilizados en el algoritmo evolutivo. Existe una diferencia sustancial en los 20 parámetros que utiliza cada versión del robot. En el OPEN-R el momento en que cada pata se debía mover en el ciclo de locomoción fue fijado por medio de unos valores predeterminados, sin embargo en el ERS-110 se usaron dos parámetros para indicar el momento en que debía moverse cada pata, lo cual se encuentrá descrito en [3].

La población inicial utilizada en el proceso evolutivo de ambas versiones se crea con una distribución uniforme dentro del rango de búsqueda dado en la Figura 2.3 y en la Figura 2.6. Cada una de las fases que componen el algoritmo evolutivo (selección, cruce, mutación y evaluación) aparecen descritas en los puntos del presente proyecto fin de carrera que resumen [2] y [3].

Resultados

Los resultados, se encuentran descritos en los apartados de este documento que resumen los artículos [2] y [3].

Conclusiones

La principal conclusión de este artículo es que se ha conseguido utilizar algoritmos evolutivos para conseguir el objetivo marcado de desarrollar una manera de andar para Aibo utilizando los sensores del robot para evaluarlos, obteniendo un individuo que es capaz de andar a una velocidad de 10 metros por minuto. Como segundo objetivo se buscaba la viabilidad de ejecutar un algoritmo evolutivo en un robot real. Mientras que el tiempo necesario para desarrollar una tarea en tiempo real con un robot puede excluir su empleo en algunas situaciones, un robot que evoluciona sus comportamientos según sus acciones en el mundo real tiene el potencial de ser un desarrollo muy positivo para un robot lúdico.

2.7. Resumen del estado del arte

Para finalizar el presente capítulo sobre el estado del arte, y poder observar con mayor claridad las diferentes técnicas de inteligencia artificial y los resultados obtenidos para lo diversos robots, se presentan los Cuadros 2.1 y 2.2 que contienen un resumen sobre cada uno de los artículos comentados anteriormente.

Titulo	Genetic Programming Approach to the Construction of a Neural Network for Control of a Walking Robot	Año	1992
Autores	M. Anthony Lewis, Andrew H. Fagg y Alan Solidum		
Técnicas IA	Algoritmo genético (torneo, cruce, mutación y elitismo) y redes de neuronas		
Tipo de robot	Rodney (Robot hexápodo)		
Resultados	El robot anda de manera trípode, como muchos insectos hexápodos		

Titulo	Autonomous Evolution of Gaits with the Sony Quadruped Robot	Año	1999
Autores	G.S. Hornby, M. Fujita, S. Takamura, T. Yamamoto y O. Hanagata		
Técnicas IA	Algoritmo genético (torneo, cruce y mutación)		
Tipo de robot	Aibo: prototipo OPEN-R (Robot cuadrúpedo)		
Resultados	Supera en velocidad al andar a los algoritmos manuales. Dos maneras de andar: paso y trote, la más rápida el paso.		

Titulo	Evolving Robust Gaits with AIBO	Año	2000
Autores	G.S. Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto y M. Fujita		
Técnicas IA	Algoritmo genético (torneo, cruce y mutación)		
Tipo de robot	Aibo: versión ERS-110 (Robot cuadrúpedo)		
Resultados	Se consigue hacer andar al robot en su nueva versión. Se incluye esta manera de andar en la versión comercial.		

Cuadro 2.1: Resumen del estado del arte (I)

Titulo	Autoguiado de robots móviles mediante redes de neuronas	Año	2004
Autores	Mª Concepción Marcos, Roberto Guzmán y Rocio Alaiz		
Técnicas IA	Redes de neuronas: perceptrón simple y perceptrón multi-capa		
Tipo de robot	PICBOT3 (Robot con 3 ruedas)		
Resultados	Capaz de generalizar, simula trayectorias con distintos orígenes y metas y con entornos variados.		

Titulo	Generando un agente robótico autónomo a partir de la evolución de sub-agentes simples cooperativos	Año	2004
Autores	Ricardo A. Téllez y Cecilio Angulo		
Técnicas IA	Agentes y redes de neuronas		
Tipo de robot	Robot con 3 ruedas		
Resultados	Se consigue el objetivo de orbitar alrededor de un objeto sin chocarse, mejorando los resultados de un multi-agente centralizado, incluso simulando el fallo de un sensor.		

Titulo	Autonomous Evolution of Dynamic Gaits With Two Quadruped Robots	Año	2005
Autores	G.S. Hornby, S. Takamura, T. Yamamoto y M. Fujita		
Técnicas IA	Algoritmo genético (torneo, cruce y mutación)		
Tipo de robot	Aibo: prototipo OPEN-R y versión ERS-110 (Robot cuadrúpedo)		
Resultados	Mejora los resultados de los algoritmos manuales. Se incluye la manera de caminar en la versión comercial.		

Cuadro 2.2: Resumen del estado del arte (II)

Capítulo 3

Objetivos

3.1. Introducción

Tras realizar un análisis de los diferentes estudios y experimentos plasmados en el capítulo anterior y aquellos publicados por otros autores, a lo largo de este capítulo se van fijar los distintos objetivos para el presente proyecto fin de carrera.

Como se ha podido observar, la mayor parte de los artículos publicados y los experimentos realizados tratan sobre la aplicación de redes de neuronas al control de los diferentes robots. Algunos artículos versan sobre la aplicación de otras técnicas de inteligencia artificial, pero todas ellas se apoyan o son complementos de las redes de neuronas y del entrenamiento de las mismas.

Durante la búsqueda, estudio y análisis del estado del arte anterior al presente proyecto fin de carrera, sólo se han encontrado tres referencias a la aplicación de algoritmos genéticos directamente a la programación de los controladores para el manejo de un robot. Estos tres artículos [1], [4] y [6] describen los resultados obtenidos por el equipo de desarrollo de Sony encargado de programar la manera de caminar del robot Aibo en su versión comercial. En ellos se explica que se utilizaron algoritmos genéticos únicamente, sin la intervención de otras técnicas de inteligencia artificial. Este hecho ha sido considerado importante, ya que en los objetivos iniciales del presente proyecto se contempló el dotar al robot Aibo de un controlador generado por algoritmos genéticos.

3.2. Especificación de los objetivos

A continuación se explicarán los diferentes objetivos marcados y desarrollados para el presente proyecto fin de carrera.

- **Realización de un interfaz de comunicación con el robot Aibo:** el propio robot incluye una librería de control de los diferentes motores que conforman sus articulaciones, sensores, botones... Esta libreria es compleja, por lo tanto se pretende realizar una librería reducida de estos controles y que proporcione una mayor facilidad de uso para poder se utilizada posteriormente en el resto del proyecto y en futuros desarrollos con el robot.
- **Realización de un algoritmo genético para el control del robot Aibo en un simulador:** se pretende generar un controlador para la manera de andar del robot por medio de la aplicación de un algoritmo genético. Este controlador se realizará para el controlador URBI, que emula el funcionamiento del robot Aibo en el entorno Webots.
- **Mejora de la distancia recorrida por el robot:** se realizará un análisis comparativo con el individuo inicial que se utiliza para generar la población inicial y que es el utilizado por URBI para las funciones de caminar del robot en el simulador Webots.

Capítulo 4

Análisis, diseño e implementación de un interfaz para un robot Aibo

4.1. Estado del arte

Existen actualmente diferentes lenguajes para la programación del robot Aibo, los cuales varían desde lenguajes considerados de bajo nivel hasta lenguajes llamados de script. A continuación se detallarán los aspectos más relevantes de los tres lenguajes más utilizados y se compararán para poder seleccionar cuál de ellos se ajusta más a las necesidades de la interfaz que se pretende diseñar e implementar para Aibo.

Cabe reseñar que todos estos lenguajes de programación funcionan bajo el sistema operativo Open-R. Este sistema operativo ha sido creado por Sony y viene instalado por defecto en la tarjeta de memoria del robot. Se encarga de compilar los programas realizados en cada uno de los lenguajes para posteriormente poder ser utilizados con el robot. También ofrece la posibilidad de interpretar directamente los comandos sin necesidad de crear un programa y tener que compilarlo. Para poder utilizar cada uno de los lenguajes que se describirán a continuación, el sistema operativo Open-R es modificado mínimamente para poder interpretar los comandos específicos de cada lenguaje. Estas modificaciones de Open-R se pueden descargar con cada uno de los lenguajes que se pretendan usar.

4.1.1. Open-R SDK

El Open-R SDK [8] y [9] es un entorno de desarrollo de aplicaciones basado en C++ que permite crear y compilar objetos de Open-R para ejecutarlos en Aibo de manera nativa. Es un interfaz que se encuentra entre la capa de sistema y la capa de aplicación.

Los objetos Open-R (también llamados módulos) son hilos independientes que pueden comunicarse entre sí por medio de paso de mensajes. Es importante aclarar que un objeto Open-R es implementado por un objeto C++. Existe un protocolo de comunicación para sincronizar este paso de mensajes. Los mensajes pueden ser cualquier tipo de C++ (int, float, ...) y son enviados por canales unidireccionales, es decir, para poder realizar una comunicación bidireccional entre dos objetos son necesarios dos canales. Además, cada canal sólo sirve para un tipo de datos, por lo tanto, serán necesarios tantos canales como tipos de datos se quieran enviar.

La capa de sistema de Open-R ofrece una serie de servicios: entrada/salida de sonido, entrada de imágenes, salida de datos... incluso la interfaz de la capa de aplicación, la cuál es implementada por el sistema de comunicación entre objetos. Estos servicios permiten a los objetos de la aplicación utilizar las funcionalidades del robot sin necesidad de un conocimiento detallado de los dispositivos hardware del mismo. Esta capa de sistema también proporciona un interfaz para el protocolo TCP/IP, que ofrece la posibilidad de desarrollar aplicaciones para la comunicación *wireless* con el robot.

La programación con Open-R SDK es considerada de bajo nivel y controla todos los aspectos de las ganancias de los motores de cada una de las partes móviles del robot. Además, controla todos los sensores, la cámara y los micrófonos y altavoces de Aibo.

4.1.2. R-Code SDK

R-Code [10] está enmarcado dentro de los lenguajes de script de alto nivel y permite crear fácilmente programas sencillos para el control de Aibo. Ofrece un entorno de programación más sencillo que utilizando C++. Es importante destacar que con el uso de R-Code no se podrá obtener más precisión en la realización de acciones complejas de la que se puede obtener con Open-R SDK. Es usado principalmente para la ejecución de acciones, movimientos o reacciones predefinidas.

Además, este lenguaje de script es capaz de realizar las siguientes acciones:

- Poner a Aibo en las tres posiciones básicas: sentado, tumbado y de pie.
- Hacer a Aibo andar, girar, chutar, tocar, mover la cabeza, y seguir con la mirada la pelota rosa.
- Hacer a Aibo reconocer 53 comandos verbales, 35 sonido y 48 tonos musicales.
- Ejecutar 600 contenidos (movimientos, sonidos, leds...).
- Adquirir datos sobre obstáculos y de los sensores.
- Utilizar secuencias IF y FOR.

- Ejecutar subrutinas.
- Utilizar variables de hasta 32 bits.
- Realizar operaciones aritméticas.
- Utilizar una cola y los comandos PUSH y POP.
- Conectarse por medio de una conexión *wireless* desde un ordenador.

4.1.3. Universal Real-time Behavior Interface - URBI

El Universal Real-time Behavior Interface, cuyo acrónimo es URBI [11] y [12], es un script de interfaz diseñada para trabajar sobre una plataforma cliente/servidor para poder controlar de manera remota el robot Aibo. Es un lenguaje simple de alto nivel que permite desarrollar controladores para el robot, tanto simples como complejos.

Este lenguaje ofrece bastante flexibilidad ya que es independiente del sistema operativo utilizado o del lenguaje de programación, ya sea C++, Java o Matlab. Esto es debido a que se pueden crear objetos en estos lenguajes para posteriormente incluirlos en URBI y ser utilizados, independientemente del lenguaje usado. Este concepto es lo que llaman los desarrolladores de URBI la modularidad del lenguaje.

Es muy interesante la estructura en capas que presenta el lenguaje para poder utilizarlo. Están divididas por niveles de complejidad, por lo tanto para realizar funciones básicas con el robot, no es necesario un gran estudio ni aprendizaje del lenguaje de script. Por este motivo es un lenguaje muy sencillo de utilizar, lo cual no le inhibe de poder realizar operaciones complejas en el manejo de Aibo.

Otro aspecto importante en comparación con R-Code SDK, es que URBI permite la paralelización de procesos. Es decir, se pueden realizar dos acciones a la vez con el robot de una manera simple. Se puede realizar diferentes eventos comenzando al mismo tiempo, uno después del otro, iniciar una acción inmediatamente después de iniciar otra...

Además, URBI [13] ofrece un simulador con compatibilidad para Webots[1], lo cuál le hace especialmente interesante para realizar simulaciones sin necesidad de disponer del robot físicamente. Aún tratándose de un lenguaje de script de alto nivel, ofrece la posibilidad de asignar ganancias a los motores del robot, al igual que en Open-R SDK, de tal manera que estaría situado entre Open-R SDK y R-Code SDK, en lo que a complejidad y funcionalidad se refiere.

Adicionalmente a las funcionalidades explicadas, este lenguaje dispone de:

[1]Webots es un simulador avanzado de robótica, http://www.cyberbotics.com/products/webots/

- Uso de variables y control de acceso concurrente.
- Posibilidad de agrupar objetos y generar listas.
- Utilizar sentencias IF, FOR, WHILE y LOOP.
- Utilización de eventos: de actuadores, sensores, temporales...
- Definición de funciones y clases.
- Métodos virtuales y atributos.
- Control de la cámara.
- Reproducción y grabación de sonido.
- Realizar operaciones aritméticas.
- Posibilidad de utilizar la librería de C++ *liburbi*.
- Conectarse por medio de una conexión *wireless* desde un ordenador.

4.2. Comparativa de lenguajes para Aibo

Después de haber estudiado algunas de las opciones que existen actualmente en el mercado para la programación del robot Aibo y de haber mostrado los puntos más importantes y principales características de cada uno de ellos, a lo largo de este apartado se mostrará una comparación entre ellos y se seleccionará la solución que mejor se adapte al proyecto que se pretende llevar a término.

En la Figura 4.1 se puede observar la relación existente entre la complejidad del lenguaje a utilizar y las funcionalidades que ofrece.

El lenguaje Open-R SDK, nos ofrece la mayor funcionalidad de todos, como se ha reflejado en el apartado anterior. Como contrapunto, se observa que la complejidad para utilizarlo es alta, ya que se trata un lenguaje de bajo nivel.
En el extremo opuesto se encuentra R-Code SDK, el cual ofrece una buena interfaz bastante simple y comprensible, lo cual limita de gran manera la funcionalidad de las acciones que se quieran realizar con el robot.
Por último, el lenguaje URBI presenta una interfaz que ofrece la posibilidad de gestionar la mayoría de los actuadores y sensores de manera parecida a Open-R SDK y de una manera más sencilla por medio de funciones y grupos.

Para el desarrollo de las funciones del robot que se desean implementar se busca una interfaz sencilla, de fácil uso y ampliables en el futuro, pero que permita poder manejar de una manera

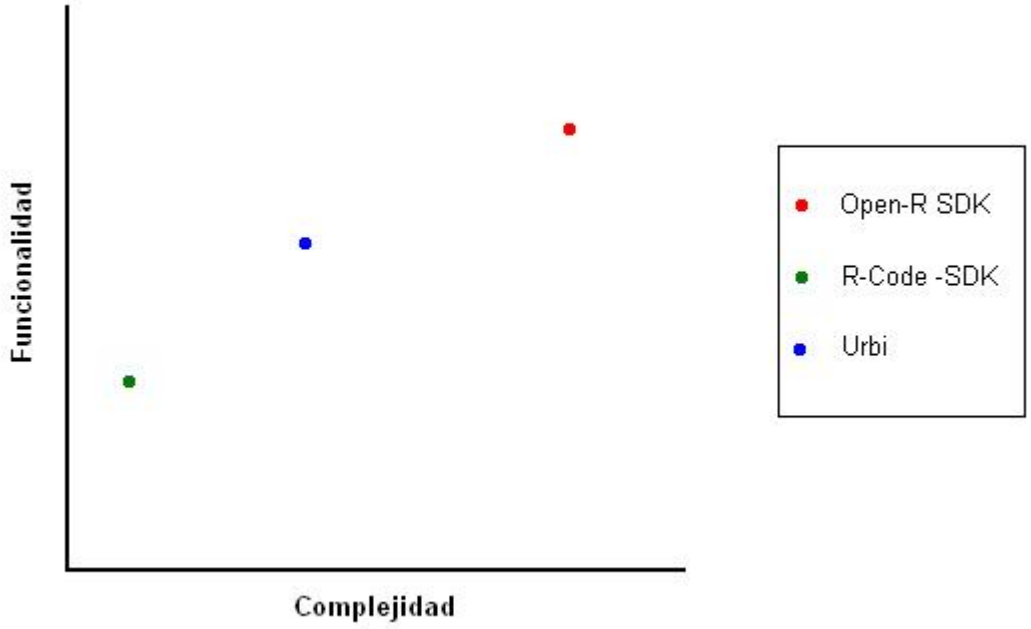

Figura 4.1: Gráfica comparativa de lenguajes para Aibo (fuente propia)

libre los motores de Aibo. Por tanto, teniendo en cuenta este aspecto, el estudio realizado de cada uno de los lenguajes y la comparativa anterior, se observa que la solución que mejor se ajusta a estos parámetros es el lenguaje de script **URBI**.

4.3. Requisitos de usuario

En este apartado se va a realizar un estudio de los requisitos de usuario para obtener posteriormente un diseño que se ajuste a los funcionalidades buscadas para el movimiento del robot. La finalidad de este apartado es conseguir unos requisitos que cumplan las siguientes normas:

- Los requisitos no deben ser ambiguos, esto quiere decir que un requisito sólo se puede entender de una sola manera.
- Los requisitos deben describir el sistema de forma completa, esto quiere decir que estos recogen la totalidad de las funciones planteadas en los requisitos de usuario y además se describe detalladamente cada punto que esté relacionado con el sistema.
- Todos los requisitos software deben poder ser verificables, esto quiere decir que a cada

requisito se le debe poder realizar una prueba para comprobar que dicho requisito se cumple.

- Los requisitos deben ser coherentes con el resto. No pueden existir requisitos que contradigan a otros requisitos.
- Cualquier requisito debe ser fácil de modificar.
- Tiene que ser fácil identificar el origen de cada requisito y que consecuencias tiene sobre el sistema.
- Los requisitos deben ser fáciles de utilizar en el resto de fases que forman parte del desarrollo del producto software.

A continuación se muestran los requisitos que debe cumplir el interfaz que se va a implementar (la situación de los motores se puede ver en las figuras 4.2 y 4.3):

1. Se podrá mover la rodilla derecha delantera indicando el ángulo y el tiempo para realizar el movimiento.
2. Se podrá mover la rodilla izquierda delantera indicando el ángulo y el tiempo para realizar el movimiento.
3. Se podrá mover la rodilla derecha trasera indicando el ángulo y el tiempo para realizar el movimiento.
4. Se podrá mover la rodilla izquierda trasera indicando el ángulo y el tiempo para realizar el movimiento.
5. Se podrá extender y encoger la cadera derecha delantera indicando el ángulo y el tiempo para realizar el movimiento.
6. Se podrá extender y encoger la cadera izquierda delantera indicando el ángulo y el tiempo para realizar el movimiento.
7. Se podrá extender y encoger la cadera derecha trasera indicando el ángulo y el tiempo para realizar el movimiento.
8. Se podrá extender y encoger la cadera izquierda trasera indicando el ángulo y el tiempo para realizar el movimiento.
9. Se podrá girar de manera circular la cadera derecha delantera indicando el ángulo y el tiempo para realizar el movimiento.
10. Se podrá girar de manera circular la cadera izquierda delantera indicando el ángulo y el tiempo para realizar el movimiento.

11. Se podrá girar de manera circular la cadera derecha trasera indicando el ángulo y el tiempo para realizar el movimiento.
12. Se podrá girar de manera circular la cadera izquierda trasera indicando el ángulo y el tiempo para realizar el movimiento.
13. Se podrá girar lateralmente la cabeza indicando el ángulo y el tiempo para realizar el movimiento.
14. Se podrá girar verticalmente la cabeza indicando el ángulo y el tiempo para realizar el movimiento.
15. Se podrá mover verticalmente la cabeza indicando el ángulo y el tiempo para realizar el movimiento.
16. Se podrán mover las orejas por separado.
17. Se podrá girar lateralmente el rabo indicando el ángulo y el tiempo para realizar el movimiento.
18. Se podrá mover verticalmente el rabo indicando el ángulo y el tiempo para realizar el movimiento.
19. Se podrá comprobar como se encuentra cada una de las patas, es decir, si se encuentran apoyadas en el suelo o no.
20. El robot podrá asentir con la cabeza.
21. El robot podrá negar con la cabeza.
22. El robot podrá ponerse de pie.
23. El robot podrá sentarse.
24. El robot podrá tumbarse.
25. El robot podrá caminar hacia adelante un número de pasos indicados.
26. El robot podrá caminar hacia atrás un número de pasos indicados.
27. El robot podrá girarse hacia la izquierda 90º.
28. El robot podrá girarse hacia la derecha 90º.
29. El robot podrá girarse 180º.
30. El robot podrá caminar lateralmente un número de pasos indicados.
31. El robot podrá caminar en diagonal un número de pasos indicados.

32. El robot podrá caminar durante un número de segundos indicados.
33. El robot podrá *dar la pata* delantera derecha.
34. El robot podrá *dar la pata* delantera izquierda.
35. El robot podrá reproducir sonidos precargados.

4.4. Diseño del sistema

Una vez finalizado el análisis del sistema y definidos los requisitos de usuario necesarios para la especificación de la interfaz que se va a realizar para el robot Aibo, se van a establecer los puntos principales para el diseño de la misma. Por tanto se generará el diagrama de clases y el diagrama de estados de la interfaz y se explicará cada uno de los componentes de ambos diagramas.

4.4.1. Diagrama de clases

Para la realización del diagrama de clases se han planteado varias formas de agrupación de funciones dependiendo de diversos factores. En un principio se pensó en la posibilidad de crear una clase para cada parte del cuerpo del robot, es decir, una clase para la cabeza y cuello, otra clase para las patas y otra clase para el rabo. Esta organización de clases sería útil para una interfaz o aplicación que creara objetos de cada uno de los elementos que conforman el robot Aibo. En un principio parece una solución válida, pero se presenta el problema de donde agrupar las funciones que combinan movimientos de diferentes partes del robot.

Como solución a esta pregunta, surgió el diseño de clases que ha sido adoptado para la realización del interfaz para Aibo. Este diseño consta de dos clases, en las cuales se agrupan las diferentes funciones dependiendo de su complejidad. Por un lado, en la clase *Simples* se enmarcan las funciones utilizadas para el manejo de cada uno de los motores del robot. En el otro lado, en la clase *Complejas* se sitúan las funciones que son combinaciones de otras funciones y que necesitan de diferentes motores para poder realizarse.

Como se observa en la Figura 4.4, existen diferentes funciones en cada una de las clases. Se va a proceder a realizar una explicación de cada una de ellas, indicando que datos deben ser introducidos en cada parámetro, los datos de retorno de las funciones y el resultado de la ejecución. Además se indicará que requisitos cumple cada función, para poder realizar una traza comprobando que todos los requisitos son plasmados en el diseño.

Dentro de la clase *Simples* se pueden encontrar las siguientes funciones:

- **mover_rodilla_del_D(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-30, 127].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la rodilla delantera derecha del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a mover la rodilla en el sentido de las agujas del reloj y en caso de ser negativo en sentido inverso a las agujas del reloj. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 1).

- **mover_rodilla_del_I(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-30, 127].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la rodilla delantera izquierda del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a mover la rodilla en el sentido de las agujas del reloj y en caso de ser negativo en sentido inverso a las agujas del reloj. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 2).

- **mover_rodilla_tras_D(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-30, 127].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la rodilla trasera derecha del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a mover la rodilla en el sentido de las agujas del reloj y en caso de ser negativo en sentido inverso a las agujas del reloj. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 3).

- **mover_rodilla_tras_I(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-30, 127].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la rodilla trasera izquierda del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a mover la rodilla en el sentido de las agujas del reloj y en caso de ser negativo en sentido inverso a las agujas del reloj. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 4).

- **extender_cadera_del_D(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-15, 93].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la cadera delantera derecha del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a extender (separar del cuerpo) la cadera y en caso de ser negativo la encogerá (juntar al cuerpo). El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 5).

- **extender_cadera_del_I(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-15, 93].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la cadera delantera izquierda del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a extender (separar del cuerpo) la cadera y en caso de ser negativo la encogerá (juntar al cuerpo). El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 6).

- **extender_cadera_tras_D(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-15, 93].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la cadera trasera derecha del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a extender (separar del cuerpo) la cadera y en caso de ser negativo la encogerá (juntar al cuerpo). El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 7).

- **extender_cadera_tras_I(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-15, 93].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la cadera trasera izquierda del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a extender (separar del cuerpo) la cadera y en caso de ser negativo la encogerá (juntar al cuerpo). El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 8).

- **girar_cadera_del_D(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-120, 134].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la cadera delantera derecha del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a girar la cadera en el sentido de las agujas del reloj y en caso de ser negativo en sentido inverso a las agujas del reloj. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 9).

- **girar_cadera_del_I(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-120, 134].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la cadera delantera izquierda del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a girar la cadera en el sentido de las agujas del reloj y en caso de ser negativo en sentido inverso a las agujas del reloj. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 10).

- **girar_cadera_tras_D(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-120, 134].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la cadera trasera derecha del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a girar la cadera en el sentido de las agujas del reloj y en caso de ser negativo en sentido inverso a las agujas del reloj. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 11).

- **girar_cadera_tras_I(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-120, 134].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la cadera trasera izquierda del robot el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a girar la cadera en el sentido de las agujas del reloj y en caso de ser negativo en sentido inverso a las agujas del reloj. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 12).

- **girar_rabo_lateral(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-59, 59].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor del rabo del robot de izquierda a derecha o de derecha a izquierda el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a girar el rabo de izquierda a derecha y en caso de ser negativo en sentido inverso. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 17).

- **girar_rabo_vertical(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [2, 63].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor del rabo del robot de arriba a abajo o de abajo a arriba el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a girar el rabo de abajo a arriba y en caso de ser negativo en sentido inverso. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 18).

- **mover_cabeza_vertical(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-79, 2].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor del cuello del robot de arriba a abajo o de abajo a arriba el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a girar el cuello de abajo a arriba y en caso de ser negativo en sentido inverso. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 15).

- **girar_cabeza_lateral(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-91, 91].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la cabeza del robot de izquierda a derecha o de derecha a izquierda el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a girar la cabeza de derecha a izquierda y en caso de ser negativo en sentido inverso. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1.000, en caso de ser menor se tomará *tiempo=1.000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 13).

- **girar_cabeza_vertical(grados,tiempo):**
  grados: es un número entero que expresa el número de grados centígrados que se debe mover el motor. El rango de grados de posición del motor es [-16, 44].
  tiempo: es un número entero que expresa el número de microsegundos que tardará en realizarse la acción. Debe ser un valor entre 1.000 y 8.000.
  Esta función mueve el motor de la cabeza del robot de arriba a abajo o de abajo a arriba el número de grados indicado en el tiempo indicado. El parámetro *grados* será un entero que en caso de ser positivo indicará el número de grados a girar la cabeza de abajo a arriba y en caso de ser negativo en sentido inverso. El parámetro *tiempo* es un entero que indica el tiempo en milisegundos que debe de tardar en recorrer los *grados*, este parámetro debe ser mayor de 1000, en caso de ser menor se tomará *tiempo=1000*. Si el número de *grados* es superior o inferior a los limites de movimiento del motor sólo se moverá el motor hasta ese límite (Requisito 14).

- **mover_oreja_D(posicion):**
  posicion: es un número natural que indica a que posición debe moverse el motor. El rango de valores es [0,1].
  Esta función mueve el motor de la oreja derecha del robot de arriba a abajo o de abajo a arriba. El parámetro *posicion* será un entero que en caso de ser 1 indicará que la oreja se quiere elevar y en caso de ser 0 indicará que la oreja se quiere bajar. En cualquier otro caso, la oreja se mantendrá en la posición en la que se encuentre (Requisito 16).

- **mover_oreja_I(posicion):**
  posicion: es un número natural que indica a que posición debe moverse el motor. El rango de valores es [0,1].
  Esta función mueve el motor de la oreja izquierda del robot de arriba a abajo o de abajo a arriba. El parámetro *posicion* será un entero que en caso de ser 1 indicará que la oreja se quiere elevar y en caso de ser 0 indicará que la oreja se quiere bajar. En cualquier otro caso, la oreja se mantendrá en la posición en la que se encuentre (Requisito 16).

- **estado_pie_del_D():** esta función comprueba si la pata delantera derecha se encuentra apoyada en el suelo o no. En caso de encontrarse apoyada devuelve 1, en caso contrario devuelve 0 (Requisito 19).
- **estado_pie_del_I():** esta función comprueba si la pata delantera izquierda se encuentra apoyada en el suelo o no. En caso de encontrarse apoyada devuelve 1, en caso contrario devuelve 0 (Requisito 19).
- **estado_pie_tras_D():** esta función comprueba si la pata trasera derecha se encuentra apoyada en el suelo o no. En caso de encontrarse apoyada devuelve 1, en caso contrario devuelve 0 (Requisito 19).
- **estado_pie_tras_I():** esta función comprueba si la pata trasera izquierda se encuentra apoyada en el suelo o no. En caso de encontrarse apoyada devuelve 1, en caso contrario devuelve 0 (Requisito 19).

En la clase *Complejos* se van a implementar las siguientes funciones:

- **encender_motores():** esta función pondrá activos los motores del robot. Si no se realiza esta acción no se podrá ejecutar ninguna de las acciones de movimiento del robot, ya que por defecto, éstos se inician apagados (Todos los requisitos).
- **apagar_motores():** esta función desactiva los motores del robot. Si se realiza esta acción no se podrá ejecutar ninguna de las acciones de movimiento del robot. Por defecto, al encender el robot los motores se encuentran desactivados (Todos los requisitos).
- **reiniciar():** esta función reinicia el robot a una posición neutral. El resultado será el robot apoyado sobre el pecho con todas las piernas estiradas, de manera que se encuentren paralelas al suelo, la cabeza estará centrada y el rabo también.
- **sentarse():** esta función moverá al robot a la posición de sentado (Requisito 23).
- **tumbarse():** esta función moverá al robot a la posición de tumbado (Requisito 24).
- **levantarse():** esta función moverá al robot a la posición de levantado sobre las 4 patas (Requisito 22).
- **andar(pasos,velocidad):**
  pasos: es un número natural que indica el número de pasos a dar por el robot.
  velocidad: es un número natural que indica la velocidad del robot. El rango de valores es [0,2].
  Esta función hace que el robot ande hacia adelante un número de *pasos* indicado. El argumento *pasos* indica la cantidad de pasos a dar (un paso está compuesto por un movimiento de cada una de las patas), con el argumento *velocidad* se indica la velocidad a la que se debe de realizar cada paso: 0 lento, 1 normal, 2 rápido. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 25).

- **andar2(pasos,velocidad):**
  pasos: es un número natural que indica el número de pasos a dar por el robot.
  velocidad: es un número natural que indica la velocidad del robot. El rango de valores es [0,2].
  Esta función hace que el robot ande hacia adelante un número de *pasos* indicado. El argumento *pasos* indica la cantidad de pasos a dar (un paso está compuesto por un movimiento de cada una de las patas), con el argumento *velocidad* se indica la velocidad a la que se debe de realizar cada paso: 0 lento, 1 normal, 2 rápido. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 25).

- **andar_por_tiempo(tiempo,velocidad):**
  tiempo: es un número natural que indica el tiempo a andar por el robot.
  velocidad: es un número natural que indica la velocidad del robot. El rango de valores es [0,2].
  Esta función hace que el robot ande hacia adelante una cantidad de *tiempo* indicado. El argumento *tiempo* indica la cantidad de tiempo a andar, el argumento *velocidad* indica la velocidad a la que se debe de realizar cada paso: 0 lento, 1 normal, 2 rápido. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 32).

- **negar():** esta función hace que el robot niegue con la cabeza. Moverá la cabeza de un lado a otro varias veces, haciendo el gesto de negación. Esta acción se puede realizar en cualquier estado (Requisito 21).

- **asentir():** esta función hace que el robot asienta con la cabeza. Moverá la cabeza de arriba a abajo varias veces, haciendo el gesto de afirmación. Esta acción se puede realizar en cualquier estado (Requisito 20).

- **reproducir_sonido(sonido):** esta función hace que el robot reproduzca por los altavoces el sonido indicado. El argumento *sonido* indica el nombre del sonido a reproducir, el cual debe estar precargado en el robot. En caso de no existir no reproducirá ningún sonido. Esta acción se puede realizar en cualquier estado (Requisito 35).

- **andar_de_lado_D(pasos,velocidad):**
  pasos: es un número natural que indica el número de pasos a dar por el robot.
  velocidad: es un número natural que indica la velocidad del robot. El rango de valores es [0,2].
  Esta función hace que el robot ande hacia el lado derecho un número de *pasos* indicado. El argumento *pasos* indica la cantidad de pasos a dar (un paso está compuesto por un movimiento de cada una de las patas) y el lado. Con el argumento *velocidad* se indica la velocidad a la que se debe de realizar cada paso: 0 lento, 1 normal, 2 rápido. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 30).

- **andar_de_lado_I(pasos,velocidad):**
  pasos: es un número natural que indica el número de pasos a dar por el robot.
  velocidad: es un número natural que indica la velocidad del robot. El rango de valores es [0,2].
  Esta función hace que el robot ande hacia el lado izquierdo un número de *pasos* indicado. El argumento *pasos* indica la cantidad de pasos a dar (un paso está compuesto por un movimiento de cada una de las patas) y el lado. Con el argumento *velocidad* se indica la velocidad a la que se debe de realizar cada paso: 0 lento, 1 normal, 2 rápido. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 30).

- **andar_hacia_atras(pasos,velocidad):**
  pasos: es un número natural que indica el número de pasos a dar por el robot.
  velocidad: es un número natural que indica la velocidad del robot. El rango de valores es [0,2].
  Esta función hace que el robot ande hacia atrás un número de *pasos* indicado. El argumento *pasos* indica la cantidad de pasos a dar (un paso está compuesto por un movimiento de cada una de las patas), con el argumento *velocidad* se indica la velocidad a la que se debe de realizar cada paso: 0 lento, 1 normal, 2 rápido. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 26).

- **andar_diagonal_I(pasos,velocidad):**
  pasos: es un número natural que indica el número de pasos a dar por el robot.
  velocidad: es un número natural que indica la velocidad del robot. El rango de valores es [0,2].
  Esta función hace que el robot ande en diagonal hacia adelante y hacia el lado izquierdo un número de *pasos* indicado. El argumento *pasos* indica la cantidad de pasos a dar (un paso está compuesto por un movimiento de cada una de las patas), con el argumento *velocidad* se indica la velocidad a la que se debe de realizar cada paso: 0 lento, 1 normal, 2 rápido. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 31).

- **andar_diagonal_D(pasos,velocidad):**
  pasos: es un número natural que indica el número de pasos a dar por el robot.
  velocidad: es un número natural que indica la velocidad del robot. El rango de valores es [0,2].
  Esta función hace que el robot ande en diagonal hacia adelante y hacia el lado derecho un número de *pasos* indicado. El argumento *pasos* indica la cantidad de pasos a dar (un paso está compuesto por un movimiento de cada una de las patas), con el argumento *velocidad* se indica la velocidad a la que se debe de realizar cada paso: 0 lento, 1 normal, 2 rápido. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 31).

- **dar_pata_D():** esta función hace al robot *dar la pata* derecha. Esta función sólo se puede realizar cuando el robot se encuentra sentado(estado *de_pie* o estado *sentado*) (Requisito 33).
- **dar_pata_I():** esta función hace al robot *dar la pata* izquierda. Esta función sólo se puede realizar cuando el robot se encuentra sentado(estado *de_pie* o estado *sentado*) (Requisito 34).
- **girar_I():** esta función hace que el robot gire hacia la izquierda 90º. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 27).
- **girar_D():** esta función hace que el robot gire hacia la derecha 90º. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 28).
- **dar_la_vuelta():** esta función hace que el robot gire 180º. Esta función sólo se puede realizar cuando el robot se encuentra levantado (estado *de_pie*) (Requisito 29).

4.4.2. Diagrama de estados

Con objeto de poder controlar y monitorizar en todo momento que posición ha adoptado el robot, se han diseñado unos estados para el mismo y un diagrama que muestra los diferentes estados y las transiciones entre estados. Hay que destacar que no todas las funciones pertenecientes a la clase *Complejas* pueden realizarse en cualquier estado. Por lo tanto, es necesario este diagrama de estados. A lo largo de este apartado se detallarán que funciones se pueden realizar en cada uno de los estados existentes.

En la Figura 4.5 se puede observar el diagrama de estados para la interfaz del robot Aibo que se va a realizar.

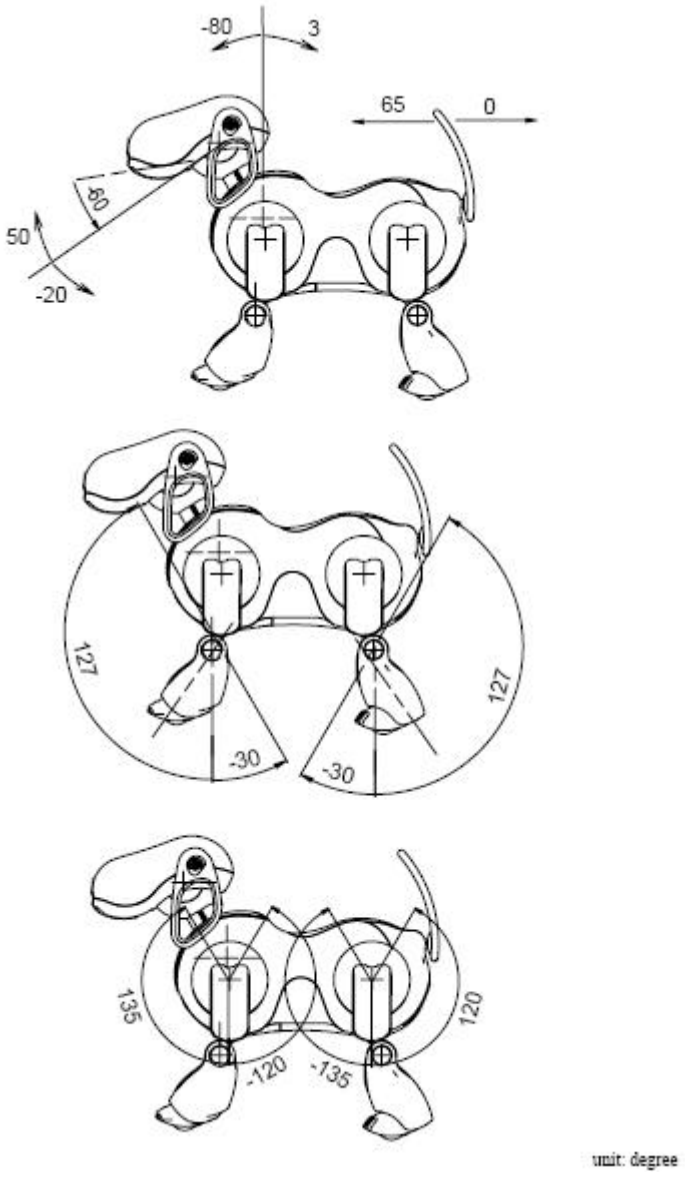

Figura 4.2: Motores del robot Aibo (vista de perfil). Fuente: Sony [7]

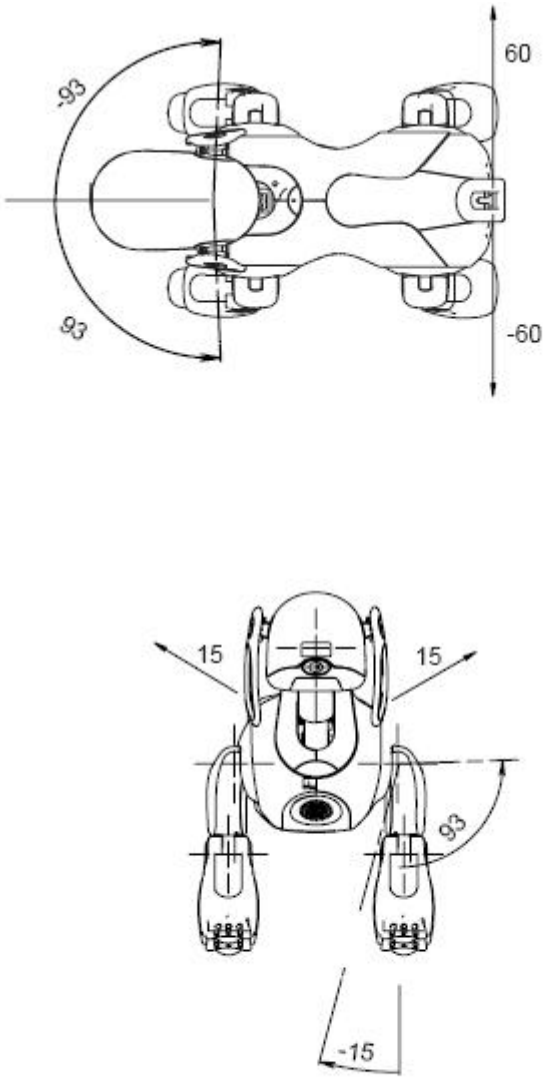

Figura 4.3: Motores del robot Aibo (vista frontal y cenital). Fuente: Sony [7]

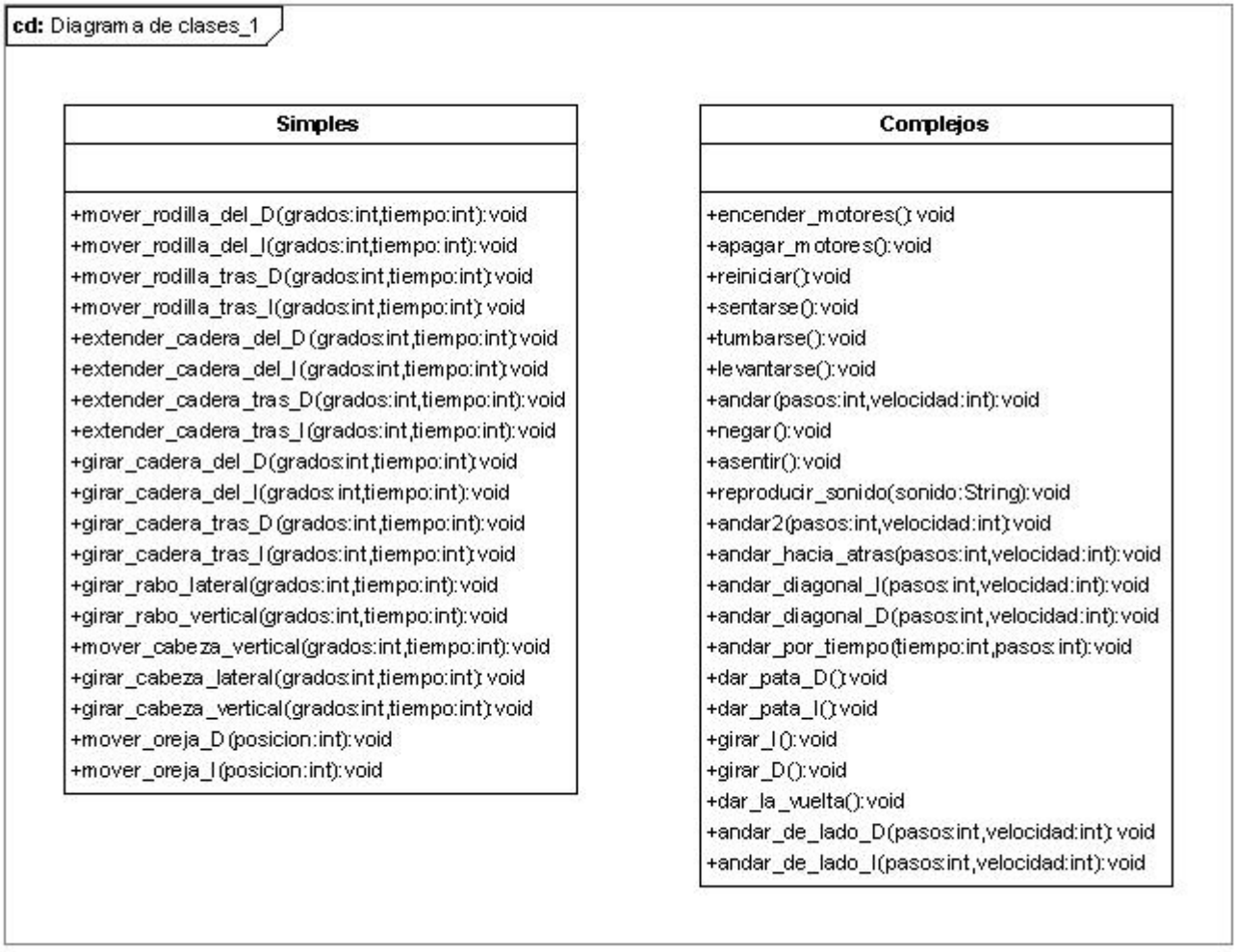

Figura 4.4: Diagrama de clases del interfaz para Aibo

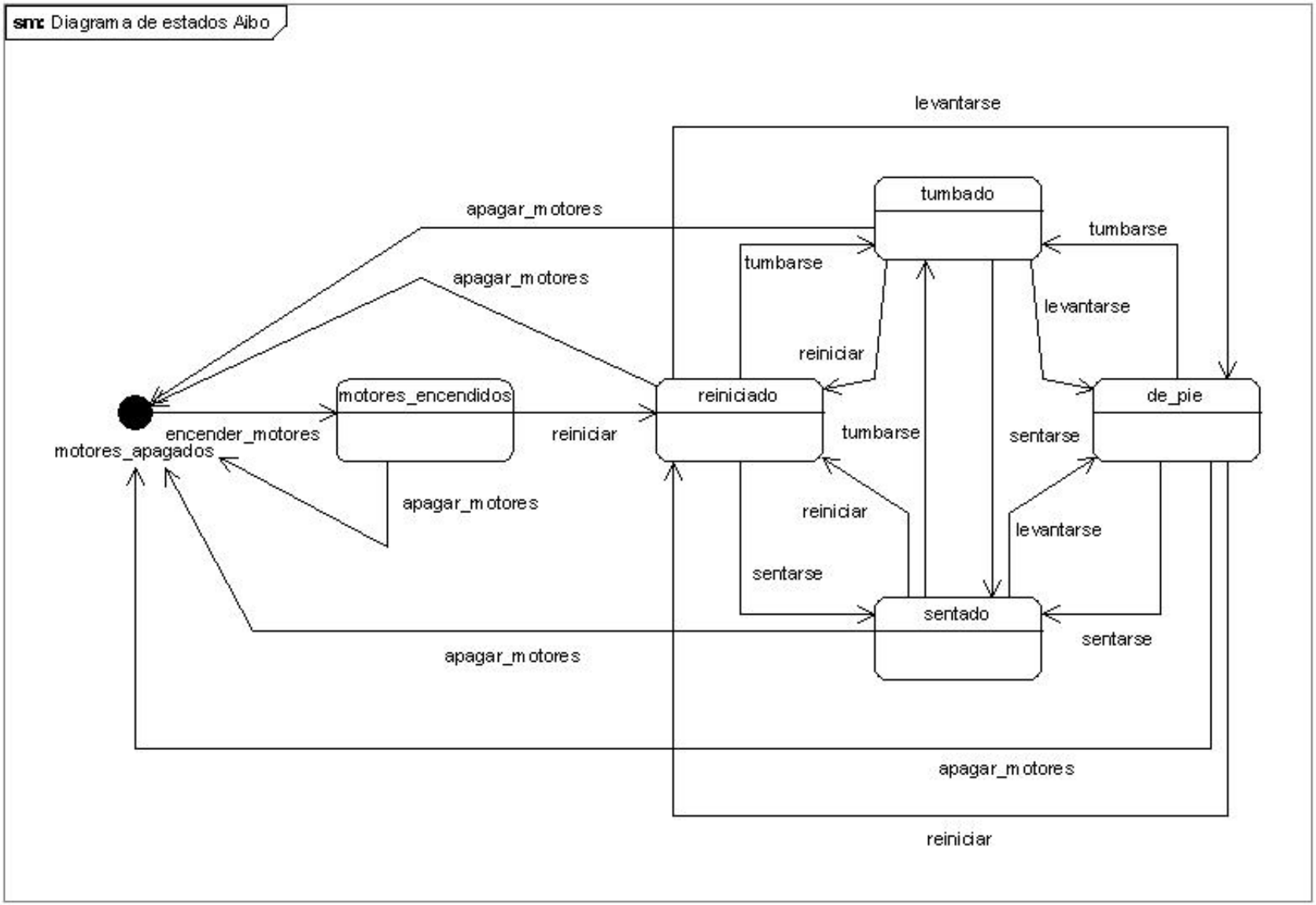

Figura 4.5: Diagrama de estados de Aibo

Existen seis estados en el diagrama de estados para el interfaz que se va a diseñar para Aibo. Cada uno de los estados tiene transiciones para pasar de un estado a otro. A continuación se describirán los diferentes estados y sus transiciones.

- **motores_apagados:** este es el estado inicial del sistema, en el que se encuentra al iniciarse el robot. También puede llegarse a este estado desde otros, como se describirá más adelante En este estado los motores están desactivados y no tienen por qué tener un valor determinado. Este estado sólo tiene una transición:
  - *encender_motores:* se realiza esta transición al ejecutarse la función *encender_motores()*. Transita del estado motores_apagados al estado motores_encendidos. El resultado en el robot es que los motores se encuentran activos para realizar movimientos.
- **motores_encendidos:** este es el estado a partir del cual se pueden empezar a realizar acciones con cada uno de los motores del robot. En este estado los motores están activados y no tienen por qué tener un valor determinado. Este estado sólo tiene dos transiciones:
  - *reiniciar:* se realiza esta transición al ejecutarse la función *reiniciar()*. Transita del estado motores_encendidos al estado reiniciado. El resultado en el robot es que los motores se encuentran activos para realizar movimientos y el robot se encuentra apoyado sobre el pecho con todas las patas estiradas de manera paralela al suelo.
  - *apagar_motores:* se realiza esta transición al ejecutarse la función *apagar_motores()*. Transita del estado motores_encendidos al estado motores_apagados. El resultado en el robot es que los motores se encuentran desactivados y no pueden realizar movimientos y no tienen ningún valor determinado.
- **reiniciado:** es un estado neutro que sirve para reiniciar el robot sin necesidad de apagarlo, en él el robot se encuentra apoyado sobre el pecho con las patas estiradas paralelas al suelo. Puede llegarse a él desde el estado motores_encendidos o desde cualquier estado de *"posición"* del robot. Este estado tiene varias transiciones:
  - *tumbarse:* se realiza esta transición al ejecutarse la función *tumbarse()*. Transita del estado reiniciado al estado tumbado. El resultado en el robot es que éste se coloca tumbado sobre el pecho.
  - *sentarse:* se realiza esta transición al ejecutarse la función *sentarse()*. Transita del estado reiniciado al estado sentado. El resultado en el robot es que éste se coloca sentado.
  - *levantarse:* se realiza esta transición al ejecutarse la función *levantarse()*. Transita del estado reiniciado al estado de_pie. El resultado en el robot es que éste se coloca de pie apoyado sobre las cuatro patas estiradas.
  - *apagar_motores:* se realiza esta transición al ejecutarse la función *apagar_motores()*. Transita del estado reiniciado al estado motores_apagados. El resultado en el robot

es que los motores se encuentran desactivados y no pueden realizar movimientos y no tienen ningún valor determinado.

Figura 4.6: Estado reiniciado

- **tumbado:** es el estado en que se encuentra el robot después de ejecutar correctamente la función *tumbarse()*, el robot se encuentra tumbado sobre el pecho. Puede llegarse a él desde los estados reiniciado, de_pie o sentado. Este estado tiene varias transiciones:

  - *levantarse:* se realiza esta transición al ejecutarse la función *levantarse()*. Transita del estado tumbado al estado de_pie. El resultado en el robot es que éste se coloca de pie apoyado sobre las cuatro patas estiradas.
  - *sentarse:* se realiza esta transición al ejecutarse la función *sentarse()*. Transita del estado tumbado al estado sentado. El resultado en el robot es que éste se coloca sentado.
  - *reiniciar:* se realiza esta transición al ejecutarse la función *reiniciar()*. Transita del estado tumbado al estado reiniciado. El resultado en el robot es que los motores se encuentran activos para realizar movimientos y el robot se encuentra apoyado sobre el pecho con todas las patas estiradas de manera paralela al suelo.
  - *apagar_motores:* se realiza esta transición al ejecutarse la función *apagar_motores()*. Transita del estado tumbado al estado motores_apagados. El resultado en el robot es que los motores se encuentran desactivados y no pueden realizar movimientos y no tienen ningún valor determinado.

Figura 4.7: Estado tumbado

- **sentado:** es el estado en que se encuentra el robot después de ejecutar correctamente la función *sentarse()*, el robot se encuentra sentado sobre las patas traseras y las patas delanteras apoyadas en el suelo. Puede llegarse a él desde los estados reiniciado, de_pie o tumbado. Este estado tiene varias transiciones:

  - *levantarse:* se realiza esta transición al ejecutarse la función *levantarse()*. Transita del estado sentado al estado de_pie. El resultado en el robot es que éste se coloca de pie apoyado sobre las cuatro patas estiradas.
  - *tumbarse:* se realiza esta transición al ejecutarse la función *tumbarse()*. Transita del estado sentado al estado tumbado. El resultado en el robot es que éste se coloca tumbado sobre el pecho.
  - *reiniciar:* se realiza esta transición al ejecutarse la función *reiniciar()*. Transita del estado sentado al estado reiniciado. El resultado en el robot es que los motores se encuentran activos para realizar movimientos y el robot se encuentra apoyado sobre el pecho con todas las patas estiradas de manera paralela al suelo.
  - *apagar_motores:* se realiza esta transición al ejecutarse la función *apagar_motores()*. Transita del estado sentado al estado motores_apagados. El resultado en el robot es que los motores se encuentran desactivados y no pueden realizar movimientos y no tienen ningún valor determinado.

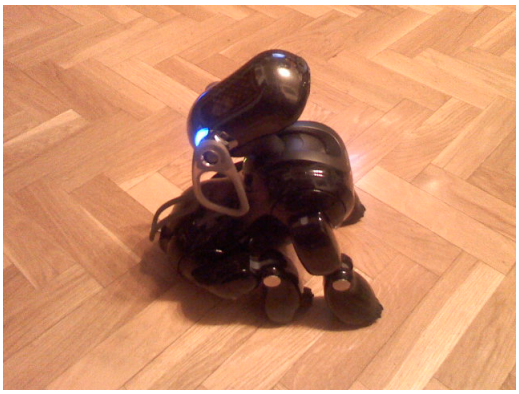

Figura 4.8: Estado sentado

- **de_pie:** es el estado en que se encuentra el robot después de ejecutar correctamente la función *levantarse()*, el robot se encuentra de pie sobre las cuatro patas estiradas y apoyadas en el suelo. Puede llegarse a él desde los estados reiniciado, sentado o tumbado. Este estado tiene varias transiciones:
  - *sentarse:* se realiza esta transición al ejecutarse la función *sentarse()*. Transita del estado de_pie al estado sentado. El resultado en el robot es que éste se coloca sentado.
  - *tumbarse:* se realiza esta transición al ejecutarse la función *tumbarse()*. Transita del estado de_pie al estado tumbado. El resultado en el robot es que éste se coloca tumbado sobre el pecho.
  - *reiniciar:* se realiza esta transición al ejecutarse la función *reiniciar()*. Transita del estado de_pie al estado reiniciado. El resultado en el robot es que los motores se encuentran activos para realizar movimientos y el robot se encuentra apoyado sobre el pecho con todas las patas estiradas de manera paralela al suelo.
  - *apagar_motores:* se realiza esta transición al ejecutarse la función *apagar_motores()*. Transita del estado de_pie al estado motores_apagados. El resultado en el robot es que los motores se encuentran desactivados y no pueden realizar movimientos y no tienen ningún valor determinado.

Figura 4.9: Estado de_pie

Por último, se muestran que funciones pertenecientes a la clase *Complejos* pueden ser ejecutadas en cada estado, ya que no todas las funciones pueden ser usadas en todos los estados.

- **motores_apagados:** encender_motores
- **motores_encendidos:** apagar_motores, reiniciar
- **reiniciado:** apagar_motores, sentarse, tumbarse, levantarse, reproducir_sonido
- **tumbado:** apagar_motores, reiniciar, sentarse, levantarse, reproducir_sonido, negar, asentir
- **sentado:** apagar_motores, reiniciar, tumbarse, levantarse, reproducir_sonido, negar, asentir, dar_pata_D, dar_pata_I
- **de_pie:** apagar_motores, reiniciar, tumbarse, sentarse, reproducir_sonido, negar, asentir, andar, andar_de_lado, andar_hacia_atras, andar_diagonal_D, andar_diagonal_I, andar_agachado, girar_D, girar_I, dar_la_vuelta

4.5. Pruebas del sistema

4.5.1. Definición del alcance de las pruebas

Con las pruebas de las funciones que se han presentado en el análisis y en el diseño de este documento, se pretende probar todas las funcionalidades del sistema. De modo que se pueda asegurar la calidad del mismo antes de su paso a lo que en un producto software correspondería

a la fase de implantación del sistema. Es decir, a su utilización por parte del usuario o usuarios finales.

Las pruebas deben recoger todos los requisitos especificados en la fase de análisis del sistema de este documento, de modo que todos sean probados y se compruebe que su funcionalidad es la esperada.

Se van a especificar cuales son los requisitos que deben cumplir las pruebas a realizar sobre el sistema. Es decir, se especificarán cuales son las pruebas a realizar, los resultados esperados de cada prueba y a que requisito de usuario afecta.

4.5.2. Definición de las pruebas de aceptación del sistema

A continuación se van a especificar las distintas pruebas a realizar sobre el sistema para comprobar su correcto funcionamiento.

La principal finalidad de estas pruebas es que se verifique como el sistema funciona correctamente antes de su implantación y comienzo de la explotación del mismo por parte de los usuarios finales.

Las pruebas a realizar serán las siguientes.

1. Comprobar que se realiza correctamente el movimiento de todos y cada uno de los motores (Figura 4.2 y Figura 4.3) que componen el robot Aibo. Se deberá realizar el movimiento de cada uno de los motores dos veces: una con un valor positivo de grados a mover y otra con un valor negativo. El valor del tiempo puede ser cualquiera superior a 1.000. (Requisitos del 1 al 18)

2. Comprobar que no se superará el rango mínimo ni el rango máximo de movimiento de los motores que componen el robot Aibo. Se deberá realizar el movimiento de cada uno de los motores dos veces: una con el máximo valor de giro de cada motor y otro con el mínimo valor de giro. También se realizará la prueba con cada una de las funciones de la clase *Complejos*. (Todos los requisitos)

3. Comprobar que no se apaga el robot debido a errores de los movimientos de los motores. Se realizarán pruebas con todas las funciones de ambas clases (Figura 4.4), usando los valores mínimos, máximos e intermedios para cada función. (Todos los requisitos)

4. Comprobar que las funciones que devuelven el estado de los sensores de los pies del robot funcionan correctamente. Se probará cada función estando el pie apoyado y posteriormente sin apoyar. (Requisito 19)

5. Comprobar que se realizan correctamente las transiciones entre los diferentes estados del robot, con arreglo a lo mostrado en la Figura 4.5. (Requisitos del 22 al 24)

6. Comprobar que no se realiza ninguna transición entre los diferentes estados del robot que no aparezca en la Figura 4.5. (Requisitos del 22 al 24)

7. Comprobar que en cada estado se pueden realizar las funciones indicadas para cada uno de ellos en el apartado del *Diagrama de Estados*. (Requisitos del 20 al 36)

8. Comprobar que no se pueden realizar funciones diferentes a las que aparecen en el apartado *Diagrama de estados* para cada uno de los estados posibles. (Requisitos del 20 al 36)

9. Comprobar que se ejecutan correctamente y tienen el resultado deseado cada una de las funciones que aparecen en la Figura 4.4. Tanto para la clase Simples, como para la clase Complejos. (Todos los requisitos)

4.6. Manual de instalación

A lo largo de este capítulo se explicará el proceso de instalación del sistema URBI, la instalación de la interfaz para el robot Aibo y la conexión y ejecución de las funciones de la misma mediante el programa Aibo Telecommande y telnet.

4.6.1. Instalación de URBI y del interfaz

Para realizar la instalación de URBI será necesaria una tarjeta de memoria programable para Aibo del tipo *memory stick* y descargar el paquete con los archivos de URBI, disponible en su página web[2].

Una vez descargado el paquete (el correspondiente al modelo de Aibo ERS7 en este caso) habrá que descomprimirlo, obteniendo la carpeta *stick-ERS7*. Seguidamente habrá que copiar el contenido de esta carpeta dentro del directorio raíz de la tarjeta de memoria. En este paso, ya tendremos lista la tarjeta de memoria para poder ejecutar URBI en el robot Aibo.

El siguiente paso es configurar la conexión wifi que utilizará Aibo. Para ello, en el directorio *OPEN-R/SYSTEM/CONF* se encuentra el archivo *WLANCONF.TXT*, que contiene los parámetros de configuración de la red inalámbrica de Aibo. Los parámetros de configuración son los siguientes:

- HOSTNAME: nombre del robot Aibo en la red inalámbrica.
- USE_DHCP: variable para usar DHCP al conectarse a una red. Con valor 1 utiliza DHCP, con valor 0 no. En caso de tomar valor 0 se deberán especificar los parámetros ETHER_IP, ETHER_NETMASK y IP_GATEWAY.

[2]http://www.gostai.com/download-redirect.php?b=aibo-stick&t=ers7

- ETHER_IP: dirección IP que tomará el robot en la red a la que se conecta.
- ETHER_NETMASK: máscara de subred de la red a la que se conecta.
- IP_GATEWAY: puerta de enlace de la red a la que se conecta.
- ESSID: nombre de la red inalámbrica a la que se va a conectar.
- WEPENABLE: indica si se deberá conectar a una red segura con contraseña o no. En caso de ser 0 se índica que no debe proporcionar una contraseña (red no segura), en caso de ser 1 se indica que debe proporcionar una contraseña a una red WEP[3], que se debe configurar en el parámetro WEPKEY.
- WEPKEY: contraseña de la red WEP.
- APMODE: debe tomar el valor 2.
- CHANNEL: canal de conexión de la red inalámbrica.

Para la instalación de la interfaz simplemente será necesario copiar el archivo con extensión *.u* en el directorio raiz de la tarjeta de memoria, para posteriormente poder ser utilizadas las funciones que ofrece desde Aibo Telecommande o desde telnet.

[3]Sólo se puede realizar conexiones a redes seguras de tipo WEP

4.6.2. Utilización del interfaz con Aibo Telecommande

Una vez iniciado el programa aparece una pantalla en la que se deberán introducir los datos del robot al que se va a conectar para poder controlarlo. Además, se muestran acciones generales de URBI para manejar ciertos aspectos del robot. En el campo *Name or IP of the robot* se deberá indicar el nombre del robot (parámetro HOSTNAME del archivo WLANCONF.TXT) o la dirección IP del robot (parámetro ETHER_IP), como se puede observar en la Figura 4.10.

Figura 4.10: Datos de conexión de Aibo Telecommande

Una vez pulsado el botón *Connect*, se intentará conectar al robot indicado. En caso de poder conectarse este botón se volverá de color verde, en caso contrario será de color rojo. Una vez conectado ya se podrán ejecutar los comandos o acciones que posea el robot, Figura 4.11.

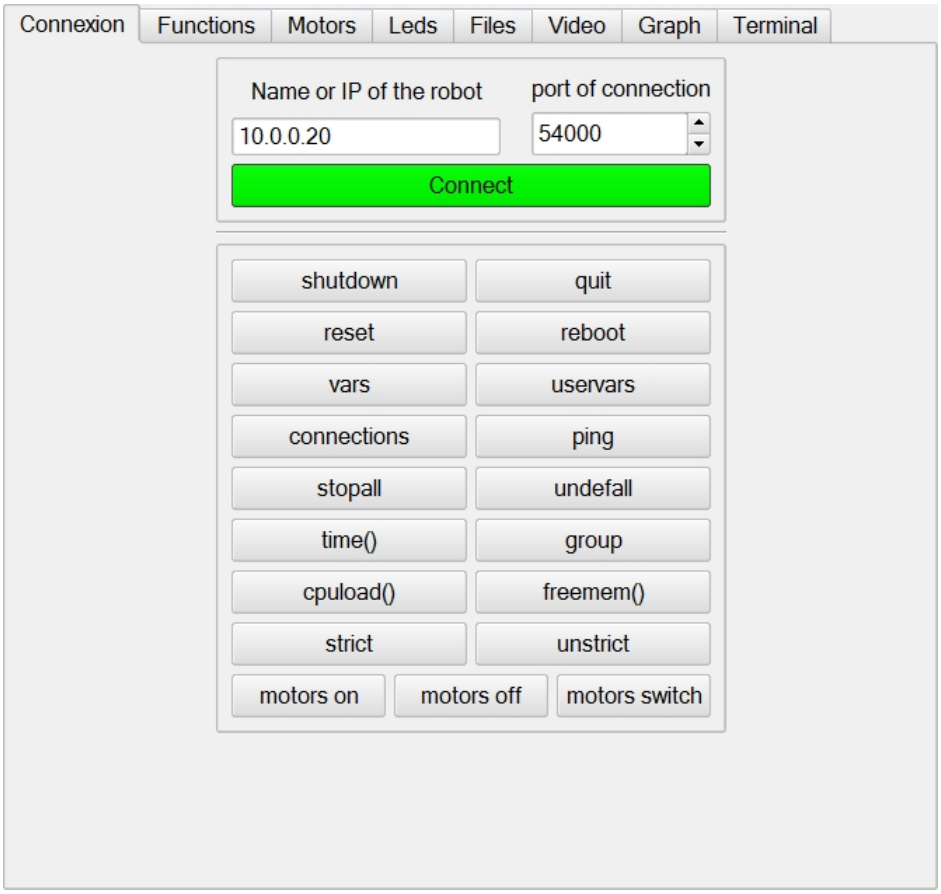

Figura 4.11: Conexión correcta de Aibo Telecommande

Cuando se conecte al robot, en la pestaña *Terminal* se mostrará el mensaje de bienvenida de Aibo con información sobre la versión de URBI, el ID del robot y el nivel de batería, Figura 4.12.

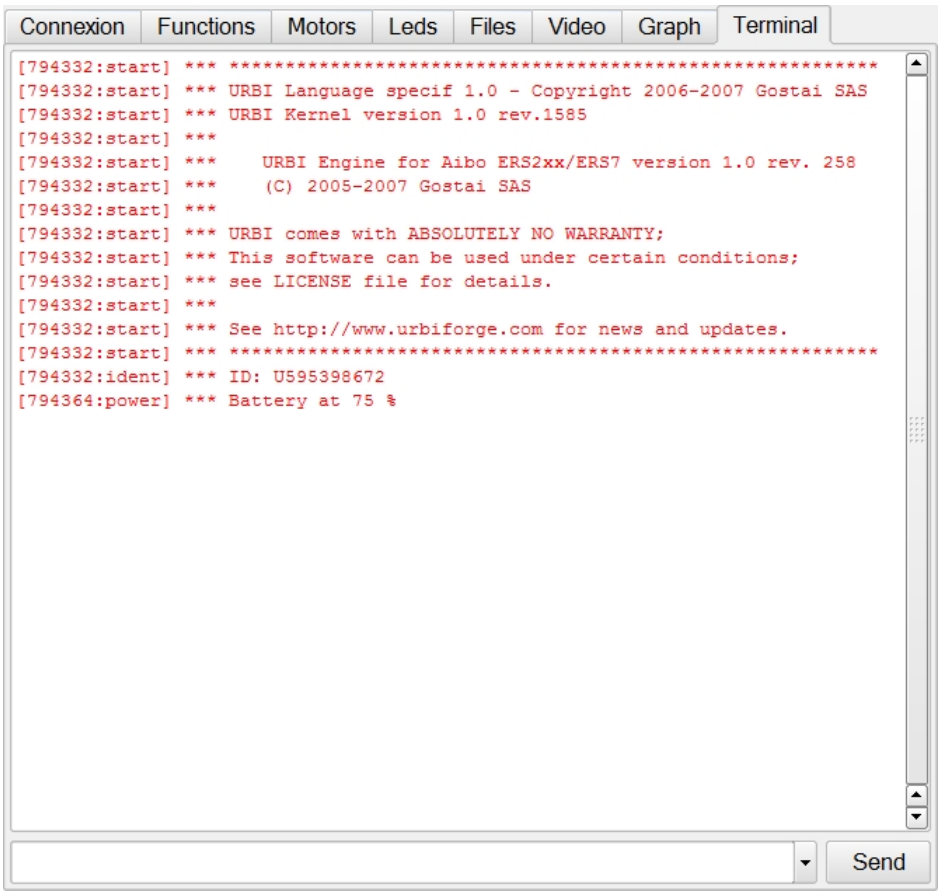

Figura 4.12: Bienvenida de conexión de Aibo Telecommande

Se deberá cargar la librería que contiene la interfaz para utilizar sus funciones (en este caso *TD.u*) usando el comando *load*[4] y pulsar el botón *Send*, Figura 4.13. En la pantalla se verá el envío del comando y si se ha ejecutado correctamente, Figura 4.14.

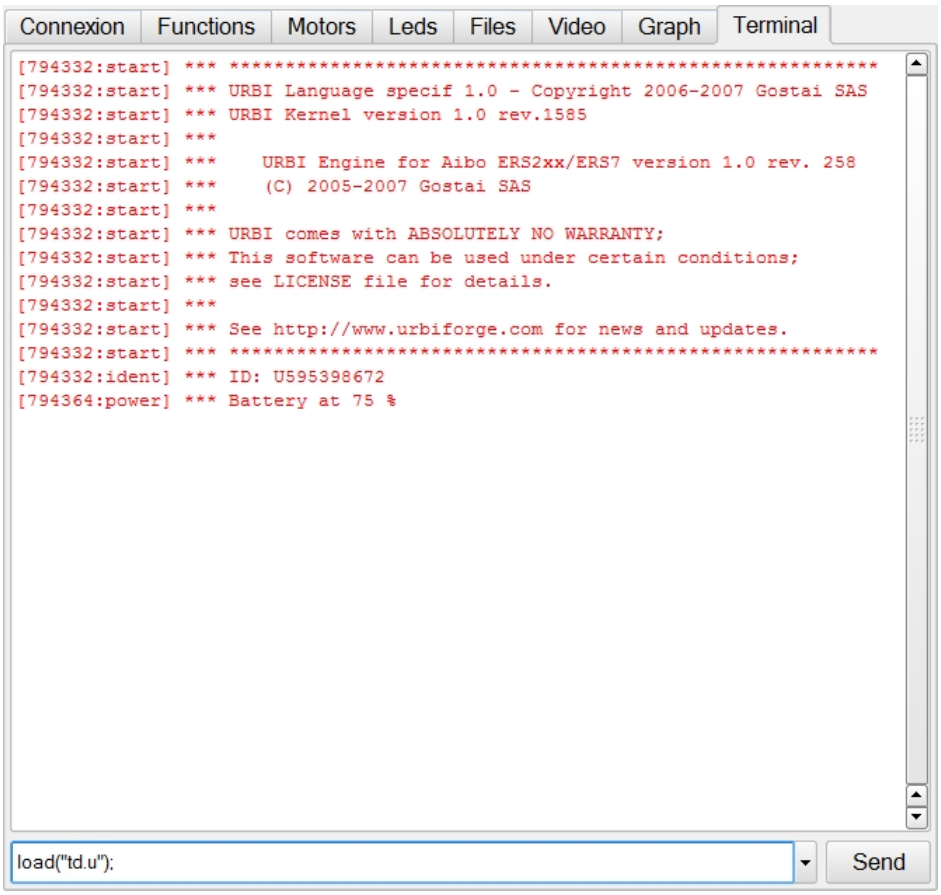

Figura 4.13: Carga de la librería de la interfaz

[4]Todos los comandos deben terminar con el símbolo *;*

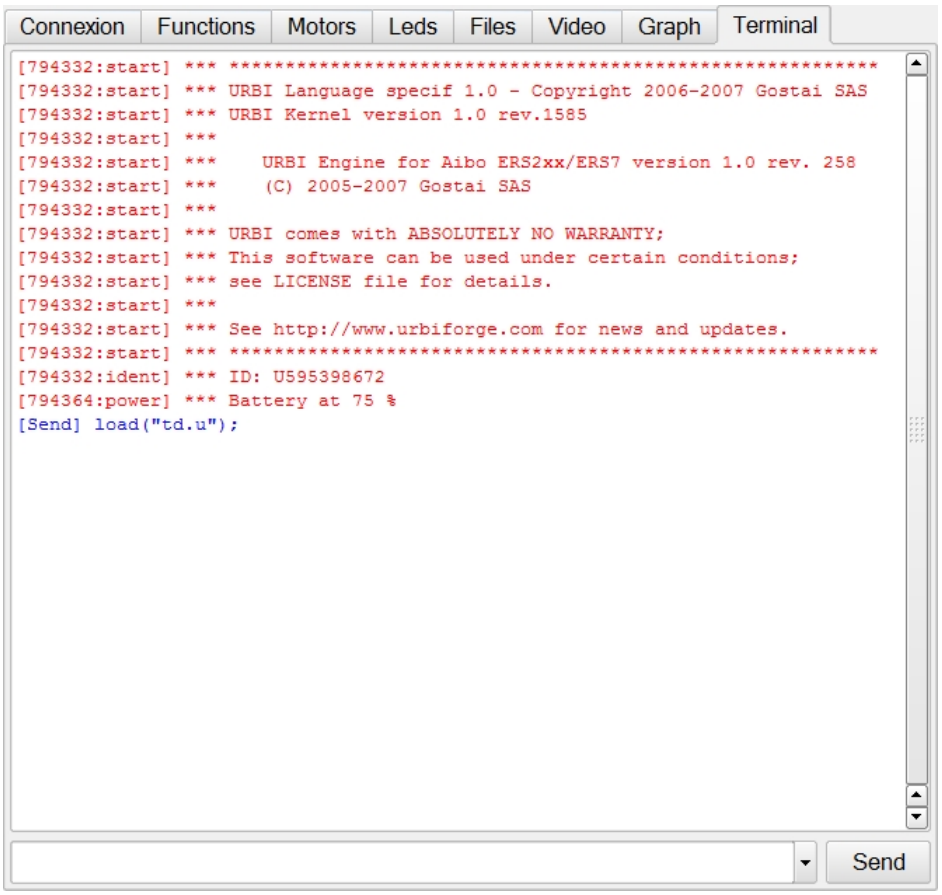

Figura 4.14: Resultado de la carga de la librería de la interfaz

Por último se introducirán las funciones a realizar por el robot, precedidas de *Simples* o *Complejos*, según pertenezcan a los movimientos simples o a los complejos respectivamente, viéndose su resultado por pantalla, Figura 4.15.

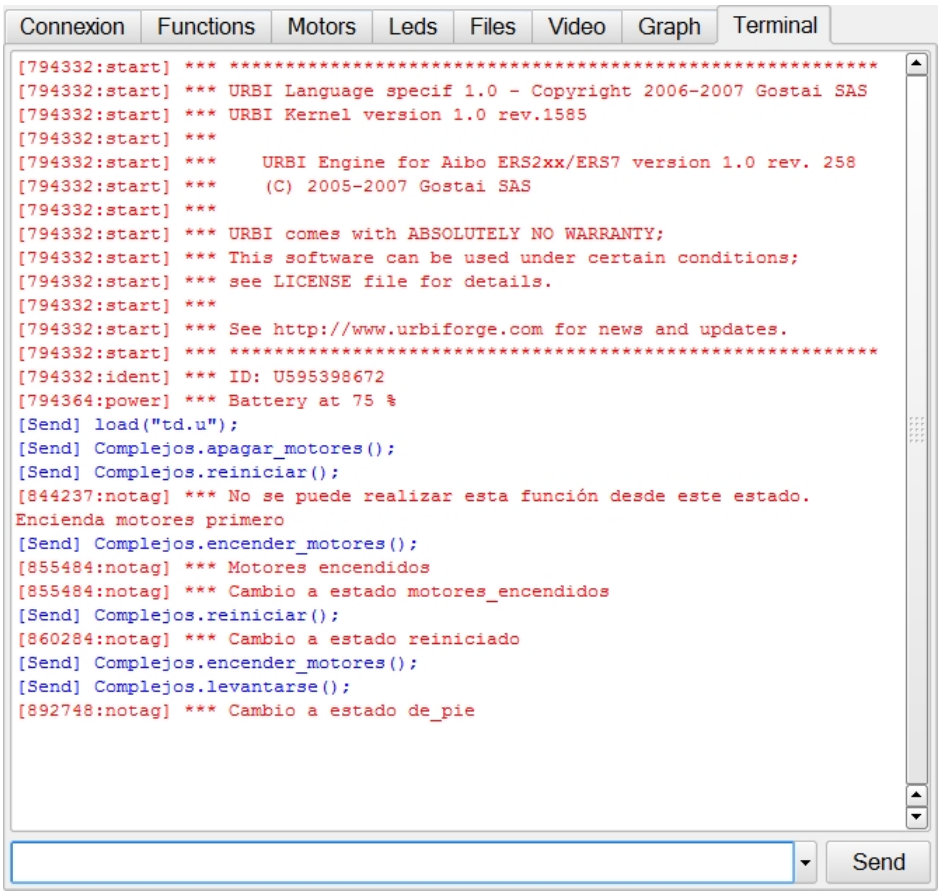

Figura 4.15: Ejecución de funciones

4.6.3. Utilización del interfaz con telnet

El sistema URBI también permite la conexión por red por medio de telnet. Para realizar la conexión será necesario abrir una conexión telnet con el nombre del robot (parámetro HOSTNAME del archivo WLANCONF.TXT) o la dirección IP del robot (parámetro ETHER_IP). Esta conexión se deberá realizar en el puerto 54.000, Figura 4.16.

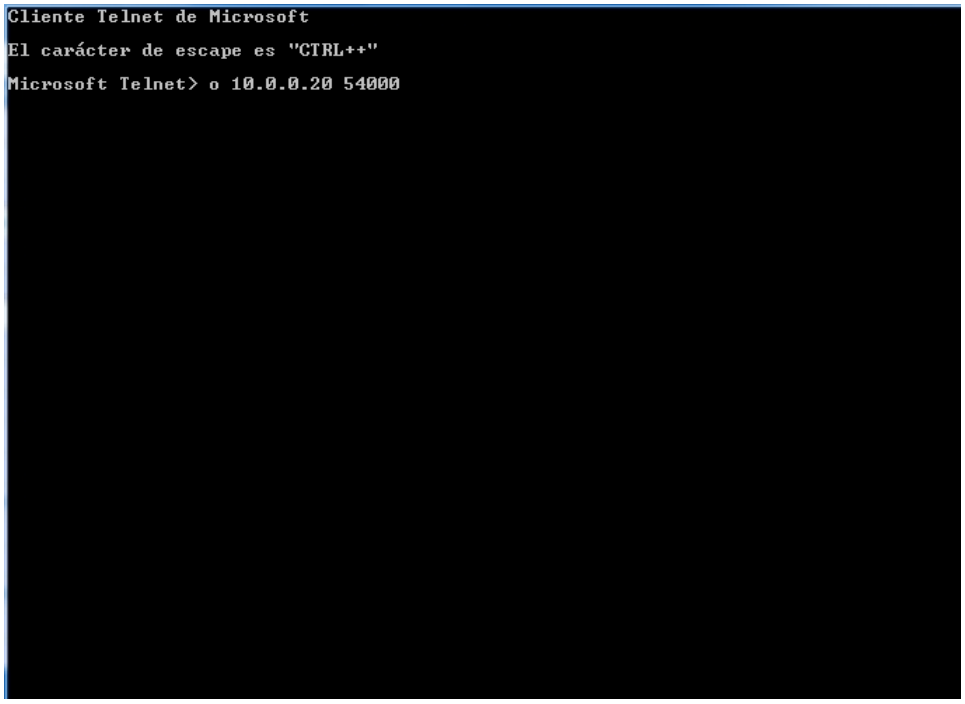

Figura 4.16: Conexión por telnet

Cuando se conecte al robot se mostrará el mensaje de bienvenida de Aibo con información sobre la versión de URBI, el ID del robot y el nivel de batería en la pantalla del cliente de telnet. El resultado es como el visto en la Figura 4.12.

Se deberá cargar la librería que contiene la interfaz para utilizar sus funciones (en este caso *TD.u*) usando el comando *load*[5]. En la pantalla se verá el envío del comando y si se ha ejecutado correctamente, similar a la observada en la Figura 4.14

Por último se introducirán las funciones a realizar por el robot, precedidas de *Simples* o *Complejos*, según pertenezcan a los movimientos simples o a los complejos respectivamente, viéndose su resultado por pantalla, similar a la observada en la Figura 4.15.

[5]Todos los comandos deben terminar con el símbolo *;*

Capítulo 5

Análisis, diseño e implementación de un algoritmo genético para el controlador de un robot Aibo

5.1. Introducción

Según John Koza, un algoritmo genético es *un algoritmo matemático altamente paralelo que transforma un conjunto de objetos matemáticos individuales con respecto al tiempo usando operaciones modeladas de acuerdo al principio Darwiniano de reproducción y supervivencia del más apto, y tras haberse presentado de forma natural una serie de operaciones genéticas de entre las que destaca la recombinación sexual. Cada uno de estos objetos matemáticos suele ser una cadena de caracteres (letras o números) de longitud fija que se ajusta al modelo de las cadenas de cromosomas, y se les asocia con una cierta función matemática que refleja su aptitud.*

Expresado de otra manera, un algoritmo genético es una técnica de búsqueda y optimización basada en la teoría de la evolución de Darwin, es decir, son métodos adaptativos que son utilizados para resolver problemas de búsqueda y optimización, basándose en el proceso genético de los seres biológicos. Usan una analogía directa con el comportamiento natural de los seres vivos. Se utiliza una población de individuos, en la cual cada uno representa una posible solución al problema planteado. A cada individuo se le valora por medio de una función, *función de fitness*, que representa la bondad de esa solución. Cuanto mayor sea la adaptación de un individuo al problema, mayor será la probabilidad de que el mismo sea seleccionado para reproducirse, cruzándose con otro individuo seleccionado de la misma manera. Este cruce produce nuevos individuos descendentes de los anteriores. De esta forma se obtiene una nueva población que sustituye a la anterior, al igual que sucede en la naturaleza con el paso del tiempo. Esto se conoce como generación. Así a lo largo de las generaciones las mejores características se pro-

pagan a través de la población, favoreciendo el cruce de los individuos mejor adaptados. Si el algoritmo genético ha sido bien diseñado, la población convergerá hacia una solución óptima del problema.

Resumiendo, un algoritmo genético consta de los siguientes pasos para cada generación, pudiendo aplicar diferentes posibles técnicas que serán explicadas posteriormente:

- **Codificación**
- **Selección**
- **Cruce**
- **Mutación**
- **Evaluación de los individuos**

5.1.1. Codificación

Se supone que los individuos (posibles soluciones del problema), pueden representarse como un conjunto de parámetros (que denominaremos genes), los cuales agrupados forman una ristra de valores, a menudo referida como cromosoma. Debe existir una representación de estos genes para poder utilizarlos posteriormente en el algoritmo genético y dotarles de unos valores. Se pueden considerar tres tipos básicos de representación o codificación de los genes:

- **Binaria:** en ella se utiliza un vector cuya longitud es la del número de genes de cada individuo y el valor que puede tomar cada elemento es un número binario.
- **Entera:** en ella se utiliza un vector cuya longitud es la del número de genes de cada individuo y el valor que puede tomar cada elemento es un número entero.
- **Real:** en ella se utiliza un vector cuya longitud es la del número de genes de cada individuo y el valor que puede tomar cada elemento es un número real.

5.1.2. Selección

En la fase de selección se realizará una elección de los individuos que pasarán a formar parte de la siguiente etapa del algoritmo genético, es decir, del cruce. Existen diferentes técnicas para realizar la selección de la población, cuyo objetivo es obtener los mejores individuos de la población para que se crucen entre ellos y así poder conseguir mejores individuos, lo cual puede producir que se llegue a una solución local y no sea la globalmente óptima.

Dentro de las diferentes técnicas de selección destacan las siguientes:

- **Selección por torneo:** en ella se escoge un determinado número de individuos de la población y se comparan sus funciones de evaluación, *fitness*, de tal manera que se selecciona al individuo que mejor se ajuste a los parámetros buscados para pasar a la siguiente fase del algoritmo genético. Esta operación se realiza el número de veces necesarias para obtener una población con el mismo número de individuos una vez que se realice el cruce.
- **Selección por ruleta:** en este proceso de selección a cada individuo se le otorga una probabilidad de ser elegido directamente proporcional a su *fitness*, de tal forma que la suma de todas las probabilidades no supere 1. Es decir, a mejor fitness mayor probabilidad de ser elegido. Posteriormente, se realiza una selección aleatoria de los individuos, en base a su probabilidad, hasta conseguir el número necesario para realizar el cruce.

Además de los diferentes métodos de selección, se pueden agregar diferentes parámetros a los mismos para favorecer a unos individuos frente a otros o para asegurar la diversidad de las diferentes generaciones. Algunos de estos parámetros pueden ser:

- **Elitismo:** al introducir elitismo en la fase de selección hace que el mejor individuo de la población actual pase a la siguiente generación, sin necesidad de ser seleccionado por la función de selección. De esta manera se asegura que el mejor individuo de la siguiente generación siempre será igual o mejor que el de la anterior. Como contrapartida, a lo largo de las generaciones puede suceder que se cree un estancamiento (nicho) de la evolución debido a que se propaga un superindividuo o a que la población no sea lo suficientemente diversa debido al elitismo.
- **Envejecimiento:** este factor se utiliza para que un mismo individuo no pueda ser seleccionado más de un determinado número de veces para realizar el cruce en diferentes generaciones, y también se puede limitar en el elitismo.

5.1.3. Cruce

Durante esta fase se cruzan o mezclan los individuos seleccionados en la fase anterior. Es decir, los genes de los dos padres se mezclan entre si para dar lugar a los diferentes hijos. Existen diversos métodos de cruce, pero los más utilizados son los siguientes:

- **Cruce basado en un punto:** los dos individuos seleccionados para jugar el papel de padres, son recombinados por medio de la selección de un punto de corte, para posteriormente intercambiar las secciones que se encuentran a la derecha de dicho punto. Es

decir, los genes del padre1 a la izquierda del punto de corte formarían parte del hijo1 y los situados a la derecha formarían parte del hijo2, mientras que con el padre2 sucedería lo contrario.

- **Cruce punto a punto:** este tipo de cruce es similar al anterior pero realizándose para cada gen de los padres. Por tanto, en este cruce los genes pares del padre1 formarían parte del hijo1 y los genes impares formarían parte del hijo2, mientras que para el padre2 sucedería lo contrario.
- **Cruce multipunto:** en este tipo de cruce se selecciona aleatoriamente o de manera fijada la cantidad de puntos que se van a utilizar para el cruce. De esta forma, y de manera análoga al anterior cruce, se irán intercambiando los genes para formar los dos nuevos hijos.

5.1.4. Mutación

La mutación se considera un operador básico, que proporciona un pequeño elemento de aleatoridad en los individuos de la población. Si bien se admite que el operador de cruce es el responsable de efectuar la búsqueda a lo largo del espacio de posibles soluciones, él operador de mutación es el responsable del aumento o reducción del espacio de búsqueda dentro del algoritmo genético y del fomento de la variabilidad genética de los individuos de la población. Existen varios métodos para aplicar la mutación a los individuos de una población, pero el más comúnmente utilizado es el de mutar un porcentaje de los genes totales de la población.

Este porcentaje de genes a mutar se puede seleccionar de dos maneras, de forma fija, especificando el mismo porcentaje de mutación a todas las generaciones del algoritmo genético y de forma variable, es decir, modificando el porcentaje de mutación de una generación a otra, por ejemplo reduciéndolo. De esta manera, se consigue hacer una búsqueda al principio e ir reduciéndolo en las siguientes generaciones.

5.1.5. Evaluación de individuos

Dos aspectos que resultan cruciales en el comportamiento de los algoritmos genéticos son la determinación de una adecuada función de adaptación o función de *fitness*, así como la codificación utilizada. Idealmente interesa construir funciones de *fitness* con ciertas regularidades, es decir, funciones que verifiquen que para dos individuos que se encuentren cercanos en el espacio de búsqueda, sus respectivos valores en las funciones de *fitness* sean similares. Por otra parte, una dificultad en el comportamiento del algoritmo genético puede ser la existencia de gran cantidad de óptimos locales, así como el hecho de que el óptimo global se encuentre muy aislado.

La regla general para construir una buena función de *fitness* es que ésta debe reflejar el valor del individuo de una manera real, pero en muchos problemas de optimización combinatoria, donde existen gran cantidad de restricciones, buena parte de los puntos del espacio de búsqueda representan individuos no válidos. Para este planteamiento en el que los individuos están sometidos a restricciones, se puede optar por varias soluciones. La primera sería la que se podría denominar absolutista, en la que aquellos individuos que no verifican las restricciones, no son considerados como tales, y se siguen efectuando cruces y mutaciones hasta obtener individuos válidos, reduciendo las posibilidades de que se propaguen. Otra posibilidad consiste en reconstruir aquellos individuos que no verifican las restricciones. Dicha reconstrucción suele llevarse a cabo por medio de un nuevo operador que se acostumbra a denominar reparador. Otro enfoque está basado en la penalización de la función de *fitness*. La idea general consiste en dividir la función de *fitness* del individuo por una cantidad (la penalización) que guarda relación con las restricciones que dicho individuo viola. Dicha cantidad puede simplemente tener en cuenta el número de restricciones violadas o bien el denominado costo esperado de reconstrucción, es decir, el coste asociado a la conversión de dicho individuo en otro que no viole ninguna restricción.

5.2. Diseño del algoritmo genético

Una vez plasmadas las diferentes posibilidades que se plantean para la realización de cada una de las fases que componen el algoritmo genético, se procederá a la elección y explicación de las herramientas seleccionadas para cada una de estas fases. Además, a lo largo de esta sección se establecerán los puntos principales para el diseño del mismo. A continuación se muestra el diagrama de clases del algoritmo genético.

5.2.1. Fase previa a la selección de herramientas para el algoritmo genético

En una primera fase del diseño del algoritmo genético surgen diferentes posibilidades para cada una de las fases que componen el mismo. Basándose en el principio de simplicidad computacional se va a proceder a la elección de un marco que reduzca la complejidad computacional sin perder de vista los elementos necesarios para la exploración de las soluciones más viables que permitan resolver los problemas especificados en los objetivos del presente proyecto fin de carrera.

Codificación

El primer punto a tratar es la representación o codificación de los genes para cada uno de los individuos de la población. Analizando el sistema utilizado por el controlador URBI, tanto en el simulador Webots como en el robot Aibo, para hacer caminar al robot, se puede observar que está formado por una matriz de números reales de un tamaño de 6 filas por 62 columnas. Esta

matriz es introducida en el motor de movimiento y sus valores son tomados por las diferentes articulaciones que componen las patas del robot para generar el movimiento. De esta forma, todas las articulaciones de las patas utilizan todos los valores de la matriz, siendo adquiridos cada uno en una fase diferente.

Teniendo en cuenta estos datos, se establece que el sistema de representación de los genes será un vector de 372 números reales, que almacenarán los datos de la siguiente posición de la articulación que está accediendo a ellos. Estos números reales respetarán el rango de actuación de las articulaciones de Aibo.

Población inicial

En una primera aproximación, previa a la selección definitiva de los parámetros que se utilizarán para los experimentos del presente proyecto fin de carrera, se realizarán experimentos con unas población de 10 y 20 individuos de 372 genes de tipo real.

Ante el desconocimiento previo del tiempo de evaluación para cada individuo, se ha optado por utilizar un tamaño de población reducido, pero suficientemente representativo para poder realizar la primera aproximación. Se toman dos tamaños diferentes de población para observar cual de ellos se comporta mejor en la búsqueda de la resolución de los problemas.

Para la generación de la población inicial se toma como base la matriz de movimiento que incluye el controlador URBI para hacer caminar al robot (individuo base). Esta matriz es modificada aleatoriamente para cada individuo de la siguiente manera:

- Selección de la cantidad modificadora: para cada uno de los elementos del vector de genes se toma el valor del individuo base y se le realiza una variación de grados entre 0^o y 10^o de manera aleatoria.
- Selección del signo de la cantidad modificadora: posteriormente se selecciona aleatoriamente y con la misma probabilidad el signo de la cantidad modificadora, es decir, si será positivo o negativo.
- Aplicación de la cantidad modificadora: por último se procede a modificar el valor del individuo base para esa posición del vector con la cantidad seleccionada aleatoriamente y su signo. Es decir, se modifica el valor de esa posición en un rango que va desde un mínimo de -10^o hasta un máximo de 10^o.

Con este rango de modificación se establece que existe la suficiente diversidad genética en la población inicial como para que cada individuo muestre suficientes diferencias que garanticen la progresión del algoritmo genético y no se produzcan estancamientos ni podas de las zonas de búsqueda. Asimismo se reduce el espacio de búsqueda a aquellos individuos que tienen una manera de andar efectiva, lo cual se complementará con la selección de la función de *fitness* que se explicará posteriormente.

Cabe destacar que a los individuos de la población inicial se les hará una evaluación adicional a la del resto de individuos de las demás generaciones. Esto se explicará en el siguiente apartado, una vez visto los parámetros que intervienen en la función de evaluación.

Función de fitness

Para la realización de la función objetivo o función de *fitness*, se han tenido en cuenta los objetivos del presente proyecto fin de carrera, y en consonancia con ellos, se ha diseñado esta función de tal manera que sean mejor valorados los individuos que recorran una mayor distancia euclídea en un número de pasos determinados.

La selección de una función de evaluación para el problema propuesto es complicada ya que, aunque se tiene un vector de genes que no varía durante la evaluación de cada individuo, los movimientos que se producen en el paso anterior condicionan a los movimientos producidos en el paso siguiente. Por lo tanto, una pequeña modificación en un gen entre dos individuos diferentes, puede dar una diferencia en la distancia recorrida muy significativa.

Teniendo en cuenta estos puntos se tomarán dos aspectos en cuenta a la hora de realizar la función:

- Posición de las patas al final del movimiento: tanto el simulador de Webots como el robot Aibo ofrecen la posibilidad de saber si las patas del robot se encuentran en contacto con el suelo. De esta forma se penalizará a los individuos que mantengan menos patas apoyadas en el suelo al final del movimiento. Esto se debe a que un individuo es menos estable cuanto menos patas tenga apoyadas en el suelo y esto puede provocar que el individuo caiga y sea incapaz de seguir desplazándose.

  Para evitar este supuesto se ha compuesto una tabla de valoración dependiendo de la cantidad de extremidades que se encuentren apoyadas en el suelo:

  - +0.5 si las cuatro extremidades se encuentran en contacto con el suelo. Es el máximo posible para esta parte de la función de *fitness*.
  - +0.25 si sólo se encuentran apoyadas en el suelo tres patas del robot. Se considera que el individuo aún sigue siendo estable cuando tiene una extremidad separada del suelo.
  - 0 si el individuo se encuentra apoyado con dos patas o menos. En este caso el individuo es altamente inestable y es muy probable que pueda caer o que haya caído al suelo.

- Distancia euclídea recorrida: para este cálculo se traslada al robot a una posición inicial de la cual se conocen las coordenadas y posteriormente se realizan una cantidad de pasos determinados. Se llaman *pasos* al movimiento completo de las cuatro extremidades. Una vez realizados los pasos, se consulta mediante GPS la posición del robot. Para evitar errores en la valoración de esta posición, se realizan tres consultas y se obtiene la media de las

tres como posición final de movimiento. Como último paso se suma la distancia euclídea desde la posición de salida hasta la posición de parada a la función de *fitness*.

El número de pasos a realizar varía dependiendo de si se trata de la evaluación de la población inicial o del resto de generaciones. En el caso de la población inicial se realizarán dos pasos y en el caso del resto de generaciones se realizarán cinco pasos. Esta diferencia es debida a que lo que se pretende en la población inicial es simplemente obtener individuos estables, entendiendo como estables aquellos individuos que no hayan caído al suelo o tengan un riesgo alto de hacerlo.

Por ello, en la población inicial se hacen dos evaluaciones de los individuos. En la primera evaluación de cada individuo se le hace andar dos pasos, una vez realizados se observa el resultado obtenido del estado de sus extremidades. Si este resultado es menor que 0.25, es decir, tiene menos de tres patas apoyadas en el suelo, el individuo se descarta y se genera un nuevo individuo aleatorio.

Una vez que se han obtenido el número de individuos indicados para la población inicial que son estables, es decir, que han obtenido más o igual a un 0.25 en la evaluación del estado de sus extremidades, se procede a realizar la misma evaluación que en el resto de generaciones. Es decir, se tendrá en cuenta el valor de la posición de sus extremidades y de la distancia euclídea recorrida.

Por lo tanto, y en resumen del caso general, el *fitness* de un individuo vendrá dado por la siguiente fórmula:

$$\textbf{\textit{Fitness = estado de las extremidades + distancia euclídea recorrida}}$$

Método de selección

Como se indicó en apartados anteriores, se pretende reducir la complejidad computacional del algoritmo genético. En base a esta premisa se ha elegido el método del torneo como operador de selección para el algoritmo genético, ya que este método es de fácil implementación, tiene una complejidad computacional reducida y garantiza en la mayoría de los casos una diversidad de individuos suficiente para poder explorar todas las ramas de las posibles soluciones al problema.

El sistema de torneo que se va a utilizar en estos experimentos previos utilizará un rango de 2 individuos para realizar el torneo. Se elegirán 2 individuos aleatoriamente de la población y se compararán los valores de sus *fitness*, el individuo que mejor fitness tenga, será incluido en la lista de individuos que formarán parte del cruce. El otro individuo será descartado, pero ambos individuos podrán volver a formar parte del torneo hasta completar el número de elementos que se utilizarán en el cruce. En caso de que el valor del *fitness* sea el mismo, se elegirá el primer individuo seleccionado de los dos.

Método de cruce

Con el grupo de individuos elegidos en la fase de selección se realizará el cruce. Para ello se tomarán los individuos de dos en dos en el orden en el que fueron seleccionados en la fase anterior, y de su cruce saldrán los nuevos individuos de la proxima generación, que reemplazarán a los padres. Por ejemplo, para la obtención del individuo 1 y del individuo 2 de la próxima generación se cruzarán el padre 1 y el padre 2 que fueron elegidos por el torneo en la fase de selección.

El sistema que se va a utilizar para el cruce de los individuos va a ser el cruce multipunto fijo. En este cruce, se establecerán de manera manual los puntos de cruce en los individuos padres. Además, se realizarán diversos experimentos con diferentes cantidades y puntos de cruce, para seleccionar aquel método que obtenga mejores soluciones, para posteriormente implementarlo en el diseño final del algoritmo genético. Para ello, se han establecido tres tipos, que son los siguientes:

- Cada gen: se realiza el cruce por cada gen. Es decir, los genes pares del padre1 formarán parte de un hijo y los impares del otro. Con el padre2 sucederá de manera análoga, pero cambiando el orden de pares e impares.
- Cada 6 genes: los intercambios de genes para generar los hijos se realizarán en bloques de 6 genes, es decir, se establecerá un punto cada 6 genes. En total 62 bloques de 6 genes.
- Cada 62 genes: los intercambios de genes para generar los hijos se realizarán en bloques de 62 genes, es decir, se establecerá un punto cada 62 genes. En total 6 bloques de 62 genes

La razón de utilizar estos tipos de cruces se debe a que, el cruce punto a punto es uno de los más utilizados, por lo tanto se cree conveniente el utilizarlo y observar los resultados obtenidos con el. En el caso de los bloques de 6 y 62 genes, se ha tenido en cuenta que la matriz de movimientos utilizada por el motor de movimientos de URBI es una matriz de 6 filas y 62 columnas. Por lo tanto se cree que puede dar buenos resultados hacer los puntos de cruces en un múltiplo de 372 (el número de genes del vector que forma un individuo) y con el número de filas que componen dicha matriz.

En el caso de los cruces realizados cada 62 genes se trata de aprovechar la ventaja de que al ser una sexta parte del total del vector se consiguen movimientos más continuos con relación a los individuos padre, pero perdiendo diversidad a la hora de cruzarse con otros individuos. En el caso de los cruces cada 6 genes se gana en diversidad genética, al producirse mayor número de cruces, pero se pierde en la continuidad de los aspectos positivos de los padres.

Método de mutación

En esta primera aproximación a los resultados ofrecidos por el algoritmo genético, y para no alterar en exceso los mismos y poder evaluar la efectividad de las herramientas seleccionadas, se ha aplicado un factor de mutación muy pequeño.

Una vez realizado el cruce de todos los individuos y obtenidos los hijos, se realizará una mutación comprendida entre -10º y 10º en 5 genes del total de la población. También se pretende observar posteriormente, cuando se haya diseñado el algoritmo genético definitivo, la mejora que supone la aplicación de la mutación en la búsqueda de la mejor solución y en la media obtenida por la población.

Resultados

Con intención de seleccionar la configuración del algoritmo genético que mejor se ajuste a las necesidades del problema y que previsiblemente mejor se comporte posteriormente para la obtención de resultados, se han realizado una serie de experimentos preliminares con las configuraciones mencionadas anteriormente para cada una de las fases del algoritmo evolutivo.

A continuación se muestran los datos obtenidos del *fitness* del mejor individuo y el *fitness medio* de la población para cada generación de la que se compone cada uno de los experimentos. Se especificarán los parámetros utilizados y posteriormente se hará un análisis comparativo con el objetivo de desarrollar el diseño definitivo del algoritmo genético.

Los primeros tres experimentos realizados, correspondientes a las Figuras 5.1, 5.2 y 5.3, fueron realizados con una población inicial de 10 individos durante 50 generaciones, utilizando para el cruce el tipo multipunto por gen, por cada 6 genes y por cada 62 genes respectivamente. En el Cuadro 5.1, se pueden observar los datos obtenidos de estos experimentos.

Fitness / Cruce	Mejor individuo	Medio población
Cada gen	1.653088083	0.89734505
Cada 6 genes	1.565063494	1.038807959
Cada 62 genes	1.23404305	0.906826889

Cuadro 5.1: Resultados para 10 individuos y 50 generaciones

Analizando los datos de las gráficas no se observa ninguna mejora sustancial ni del *fitness* del mejor individuo ni del medio de la población. Asimismo no se observa una pauta ascendente en el crecimiento de ambos valores, salvo en el caso del cruce cada 6 genes, donde se puede apreciar un ligero ascenso. Cabe destacar que no se puede desprender del estudio de los *fitness* ninguna pauta en los elementos de la población.

En un primer momento se valora la posibilidad de que la población inicial sea muy reducida y

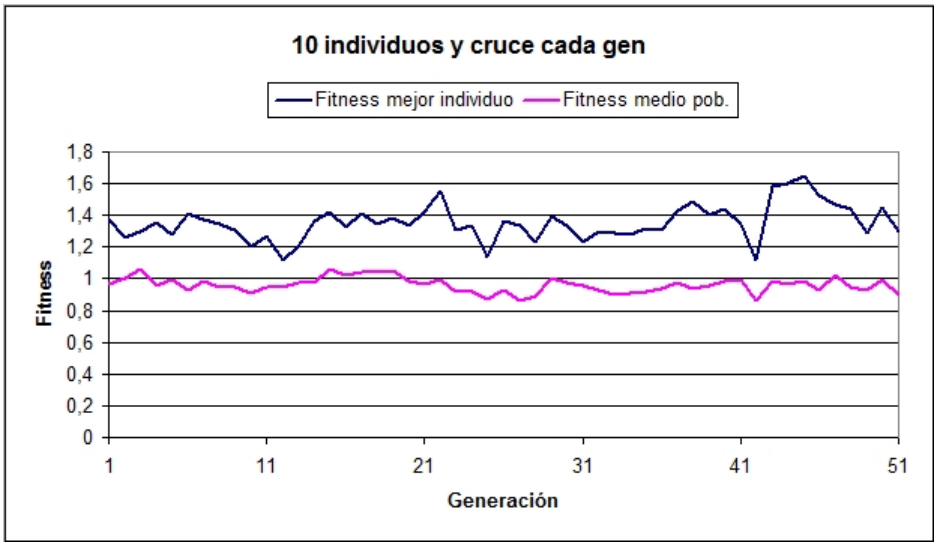

Figura 5.1: 10 individuos, 50 generaciones y cruce cada gen

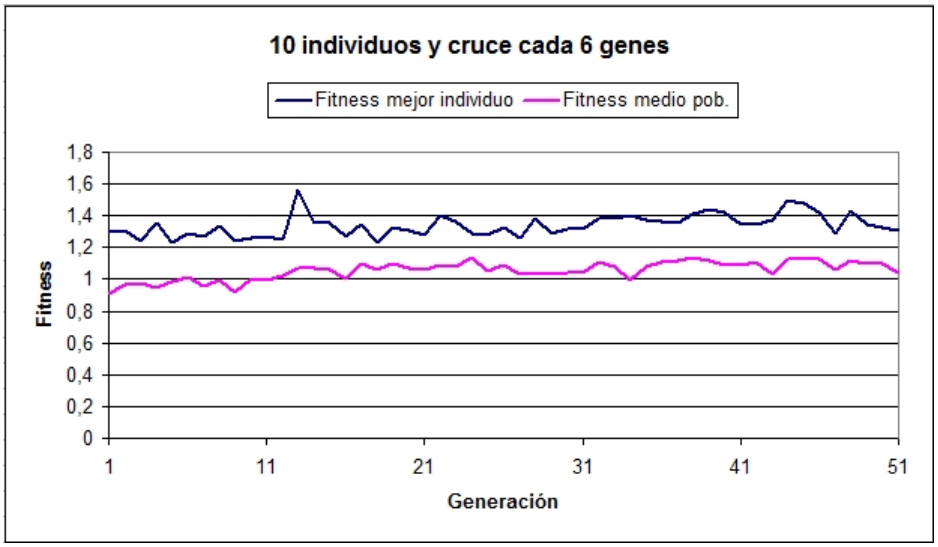

Figura 5.2: 10 individuos, 50 generaciones y cruce cada 6 genes

que el número de generaciones sea insuficiente para conseguir resultados favorables. Se opta por aumentar el número de generaciones del experimento en vez del número de individuos. Esto es debido a que al contener 372 genes de tipo real, generados aleatoriamente, para cada individuo, se estima que se mantiene la diversidad poblacional y se opta por desarrollar la otra vía.

Para realizar los próximos experimentos se aumenta el número de generaciones del algoritmo genético a 100 y 150, y se realizarán para los experimentos en los cuales no se había observado una tendencia ascendente en el *fitness*, es decir, para los casos del cruce cada gen y el cruce cada 62 genes. Observando los siguientes resultados en las Figuras 5.4 y 5.5 para 100 generaciones

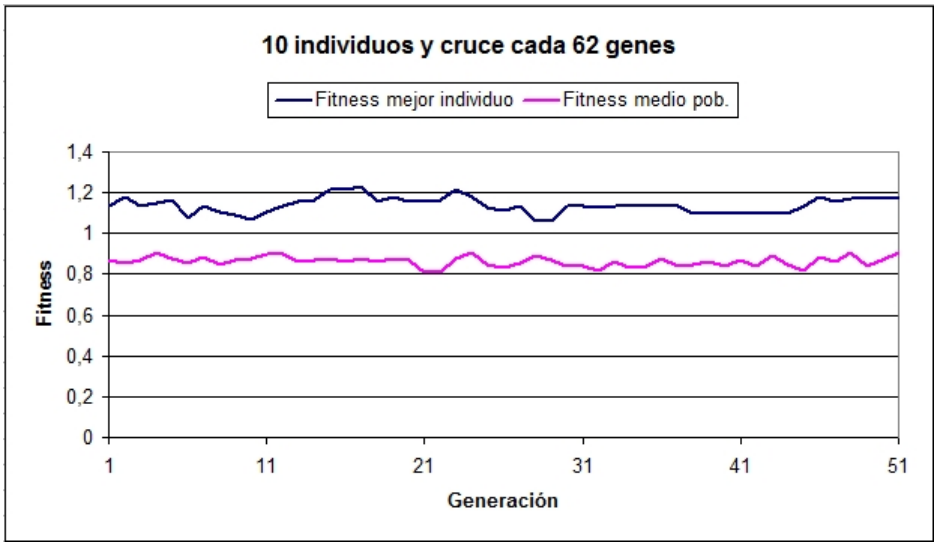

Figura 5.3: 10 individuos, 50 generaciones y cruce cada 62 genes

y en las figuras 5.6 y 5.7 para 150 generaciones.

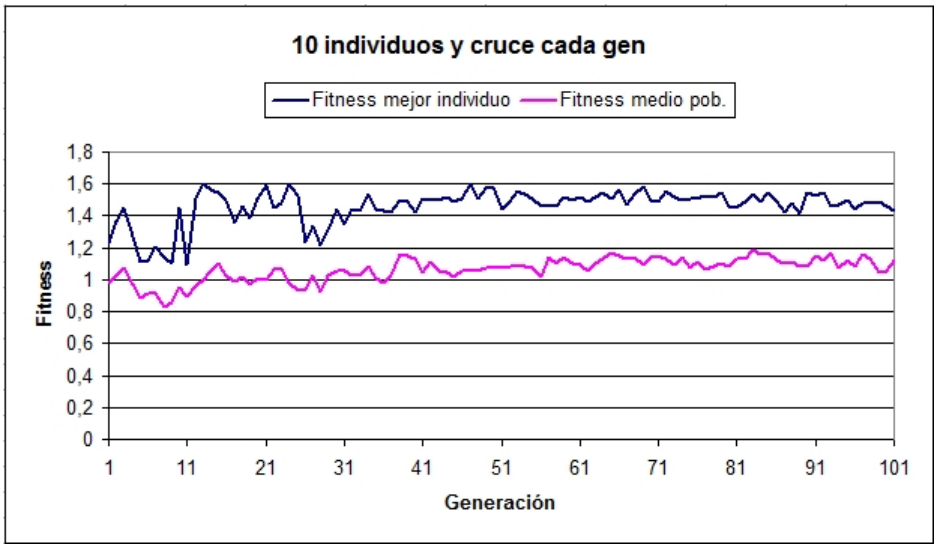

Figura 5.4: 10 individuos, 100 generaciones y cruce cada gen

Los datos obtenidos para 100 generaciones se pueden contrastar en el Cuadro 5.2, en el cual se muestran el *fitness* del mejor individuo obtenido en cada caso y del medio de la población.

En los casos en los que se utilizaron 100 generaciones no se observa una evolución en ninguno de los *fitness* recogidos durante la ejecución. Asimismo, en los experimentos para 150 generaciones que se observan en las Figuras 5.6 y 5.7 y los datos del Cuadro 5.3 tampoco se observan mejoras en comparación con las ejecuciones con 50 individuos.

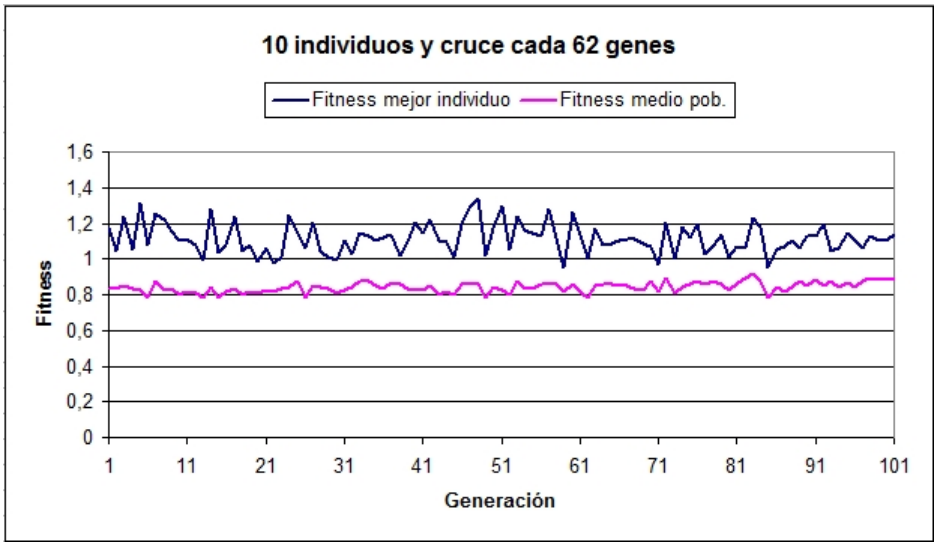

Figura 5.5: 10 individuos, 100 generaciones y cruce cada 62 genes

Fitness / Cruce	Mejor individuo	Medio población
Cada gen	1.606110746	1.123475721
Cada 62 genes	1.341374029	0.893069368

Cuadro 5.2: Resultados para 10 individuos y 100 generaciones

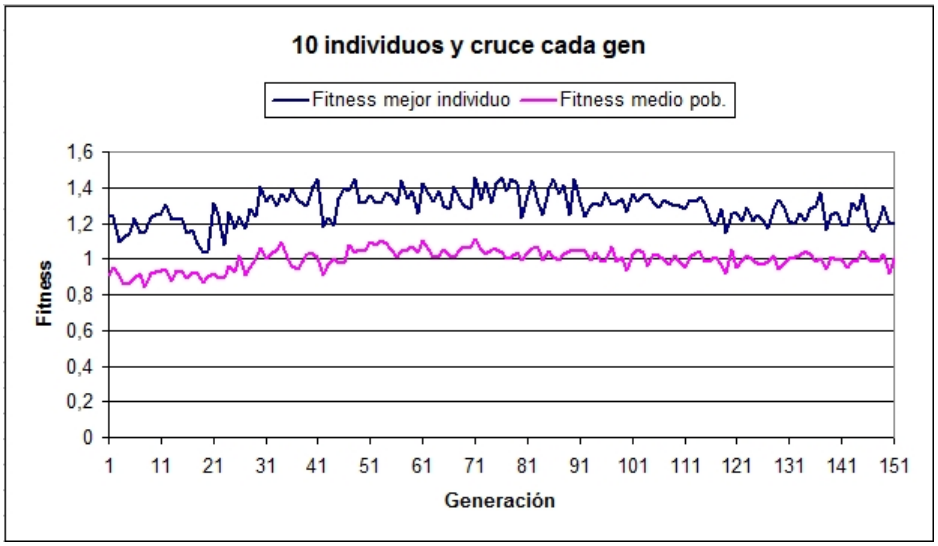

Figura 5.6: 10 individuos, 150 generaciones y cruce cada gen

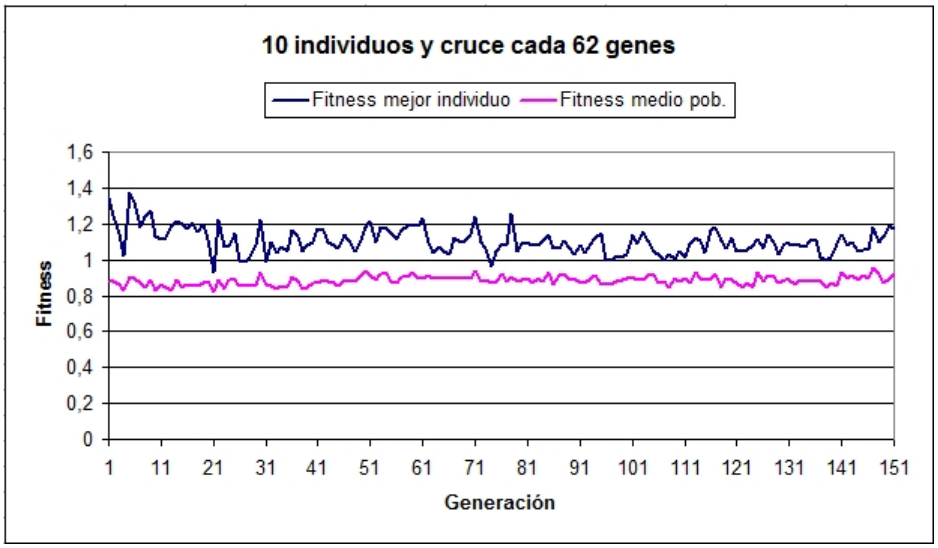

Figura 5.7: 10 individuos, 150 generaciones y cruce cada 62 genes

Fitness / Cruce	Mejor individuo	Medio población
Cada gen	1.454463613	0.99956497139552
Cada 62 genes	1.37177913	0.918223074

Cuadro 5.3: Resultados para 10 individuos y 150 generaciones

Realizados estos experimentos, y comprobando que no se ha conseguido una evolución positiva de la valoración del *fitness*, se plantea como solución a estos problemas y posible mejora de las gráficas la posibilidad de añadir elitismo en cada generación. Para ello, antes de la fase de selección se elegirá al individuo con mejor *fitness* y se utilizará como padre en la fase de cruce. De igual manera, y para dotar de diversidad genética a la pequeña población, se realizará elitismo con un individuo escogido aleatoriamente de entre todos los de la población. Como se puede observar en las Figuras 5.8, 5.9 y 5.10, las cuales son el resultado de experimentos con diferentes número de generaciones, cruces e individuos de la población pero usando elitismo, los resultados del *fitness* del mejor individuo denotan una ligera tendencia positiva, sobre todo en el cruce cada 6 genes.

El dato más significativo que se desprende en estas imágenes, es que aún usando elitismo hay generaciones en la que el mejor individuo tiene una valoración inferior a la del mejor individuo de generaciones anteriores. Teniendo en cuenta este dato, se realiza un análisis en profundidad de los resultados obtenidos para cada individuo. Especialmente de los individuos utilizados en el elitismo. Se observa que para un mismo individuo no se obtiene el mismo resultado tras dos evaluaciones del *fitness*. Esto es debido a los diferentes efectos físicos del simulador y del movimiento del robot, provocando que no siempre se obtenga el mismo resultado realizando

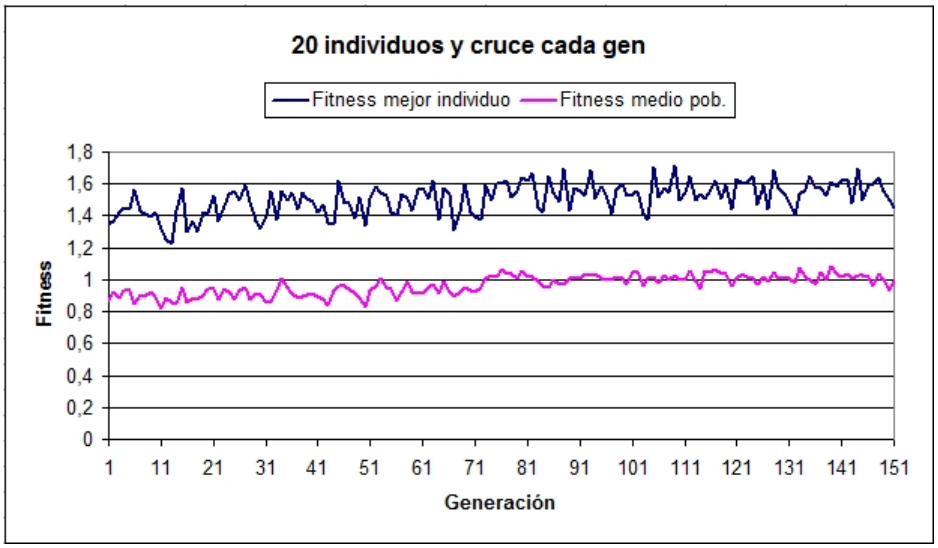

Figura 5.8: 20 individuos, 150 generaciones, cruce cada gen y elitismo

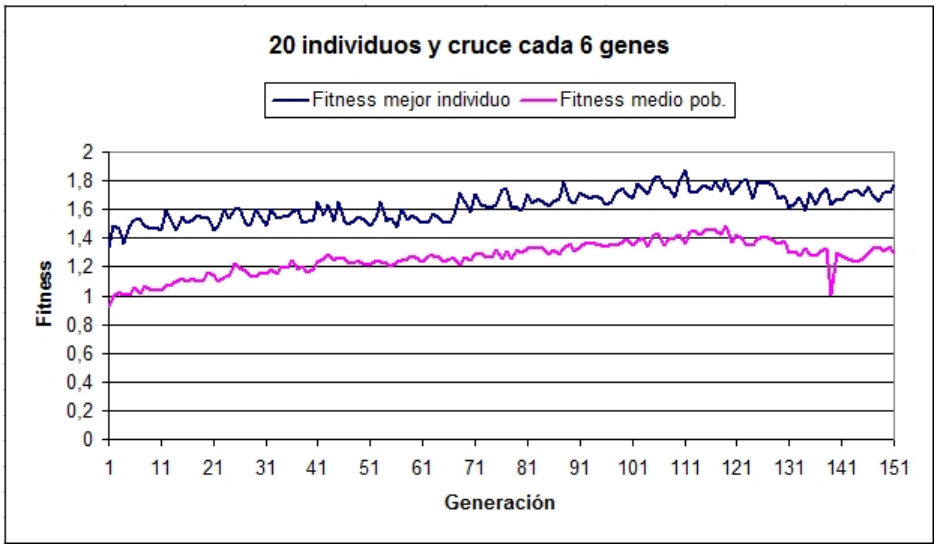

Figura 5.9: 20 individuos, 150 generaciones, cruce cada 6 genes y elitismo

los mismos movimientos, al igual que ocurre en el mundo real, ya que esto depende de muchas variables físicas y del entorno.

A lo largo de este punto se han estudiado los casos y observado la problemática de cada experimento y las ventajas que ofrece cada configuración del algoritmo genético. En el siguiente siguiente apartado se sentarán las bases de las configuraciones y herramientas a utilizar a lo largo de los experimentos que se utilizarán para el análisis de los resultados del presente proyecto fin de carrera.

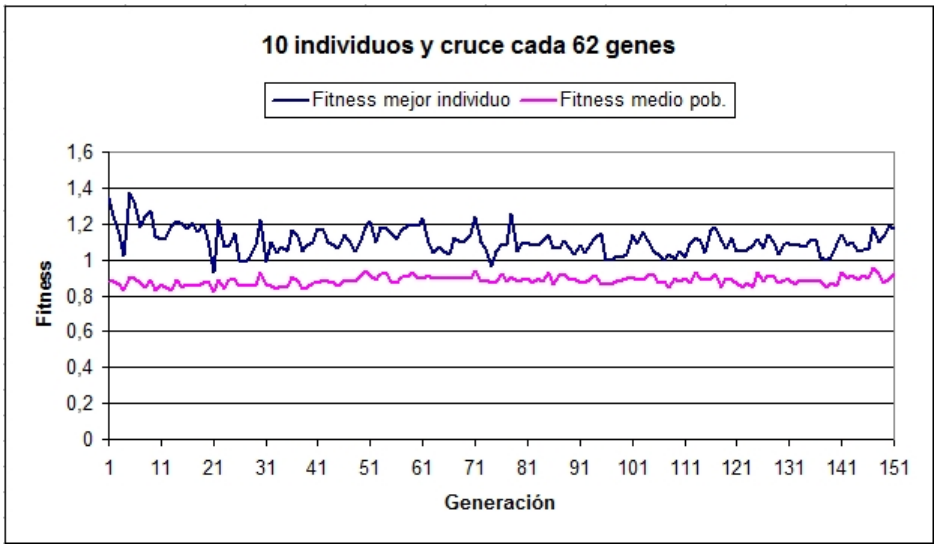

Figura 5.10: 10 individuos, 150 generaciones, cruce cada 62 genes y elitismo

5.2.2. Diseño final del algoritmo genético

Una vez realizados diferentes experimentos con diversos parámetros y herramientas para cada una de las fases del algoritmo y analizados los resultados desprendidos de ellos, se fija el marco bajo el que se van a realizar los experimentos que darán lugar a la obtención de resultados del presente proyecto fin de carrera. Este marco viene dado por el análisis de los datos de los experimentos previos y la experiencia adquirida al observar los errores y las posibles soluciones y mejoras que se han apreciado durante la ejecución de los mismos.

Población inicial

Se utilizará una población inicial de 50 individuos cuyos genes serán codificados de igual manera que los individuos que fueron utilizados para los experimentos previos. Se fija la población de cada generación en 50 individuos, ya que se estima que con ello se garantiza la diversidad genética suficiente para alcanzar soluciones óptimas a los problemas planteados. Además, con la utilización de este número de individuos, se alcanza una buena relación individuos/tiempo de ejecución.

A partir de esta cifra de individuos no se observan mejoras considerables en las soluciones obtenidas en las pruebas previas que hagan asumir un mayor coste temporal en cada ejecución del algoritmo genético.

Método de selección

En la fase de selección se utilizará el método de torneo para elegir los individuos que formarán parte de la fase de cruce del algoritmo genético. Se introduce un par de modificaciones al método

de torneo básico que son las siguientes:

- Presión selectiva: es decir, el número de individuos que se evalúan en cada ronda del torneo. Será del 10 % del total de los individuos de la población, en este caso y acorde a lo expresado anteriormente, serán 5 individuos en cada ronda del torneo los que deberán compararse y se elegirá el de mayor *fitness*.
- Elitismo: observando los problemas analizados en los resultados de los experimentos previos, se modifica el elitismo utilizado para solucionarlos. En cada generación se realizará elitismo con el mejor individuo de esa generación y con un individuo seleccionado aleatoriamente. En caso de que el mejor individuo sea el mismo que en la generación anterior y que su *fitness* sea menor, se sustituirá su *fitness* por el de la generación anterior. De esta manera se garantiza que el *fitness* del mejor individuo nunca disminuya. El motivo por el cual la valoración del mejor individuo es inferior a la del mejor individuo de la generación anterior es debida a las interactuaciones con la física del entorno explicado anteriormente. Aplicando esta modificación se garantiza que un individuo que ha conseguido obtener un buen resultado pueda continuar estando presente en la siguiente generación.

Método de cruce

Para el operador cruce se utilizará el cruce multipunto realizado cada 6 genes, puesto que en los experimentos previos realizados se han obtenido los mejores resultados cuando se utilizaba este tipo de cruce. Adicionalmente, este tipo de cruce realiza una mezcla suficiente de genes entre los individuos padre como para poder crear individuos nuevos genéticamente diferentes y diferenciados a la hora de caminar.

Método de mutación

El factor de mutación también se modifica con respecto al utilizado en las pruebas anteriores. Se ha observado que una mutación de 5 genes del total de la población es prácticamente nula y no aporta el efecto de exploración que se pretende con ella. Por ello, se establece un porcentaje base de mutación del 5 % del total de los genes de los individuos de la población. La manera de realizar la mutación es la explicada anteriormente para los experimentos previos, es decir, modificando el gen seleccionado con un valor entre -10º y 10º.

5.2.3. Diagrama de clases

Una vez finalizado el análisis de las diferentes herramientas a utilizar en el algoritmo genético y de los resultados obtenidos en la fase previa, y definidos los objetivos del presente proyecto fin de carrera, se van a establecer los puntos principales para el diseño de la misma. Por tanto, se utilizará un diagrama de clases diseñado para albergar al algoritmo genético. En la Figura 5.11, se puede observar dicho diagrama de clases que servirá para la generación del algoritmo.

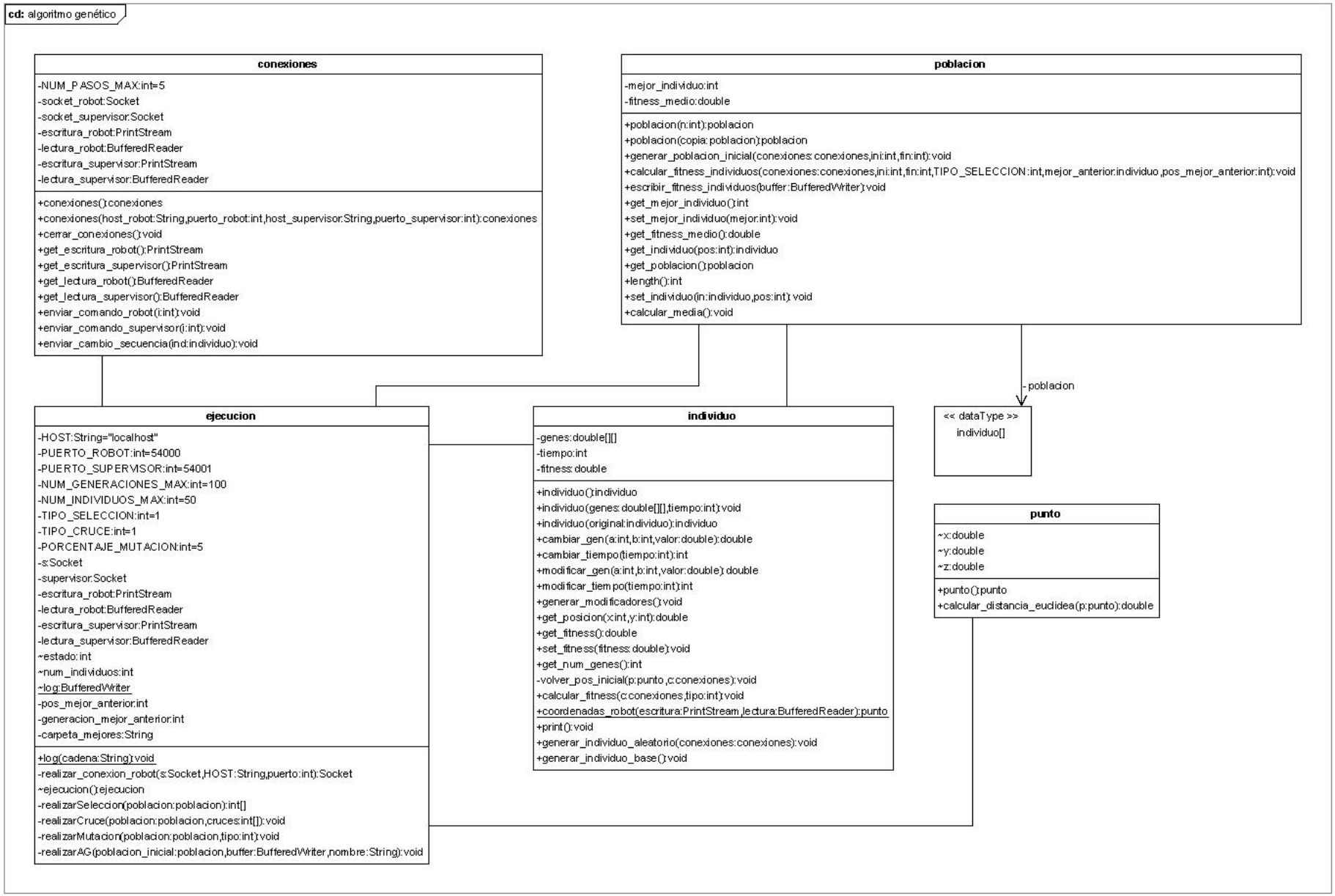

Figura 5.11: Diagrama de clases del algoritmo genético

Capítulo 6

Análisis de resultados del algoritmo genético

Para la realización del análisis de resultados que se va a realizar a lo largo del presente capítulo, se han realizado diversos experimentos sobre el simulador Webots para el controlador URBI diseñado para el robot Aibo por la empresa Gostai. Dicho simulador, en su versión de evaluación, permite tanto el control de todos los sensores y motores del robot, como de todos los parámetros del entorno que lo rodea. Al ser una versión de evaluación, posee ciertas limitaciones que se reflejarán posteriormente en el capítulo de conclusiones y líneas futuras de investigación.

A lo largo de la fase de experimentación se han realizado diversas ejecuciones con la configuración del algoritmo genético que se explicó en la sección de *"Diseño final del algoritmo genético"*. Por lo tanto la configuración queda como sigue:

- **Población inicial:** 50 individuos aleatorios con 372 genes de tipo real.
- **Generaciones:** 150 generaciones.
- **Selección:** torneo con presión selectiva del 10 % y elitismo con sustitución.
- **Cruce:** Multipunto cada 6 genes.
- **Mutación**: el 5 % del total de genes de la población.

Con esta configuración, se han realizado 30 experimentos para tratar de conseguir los objetivos del presente proyecto fin de carrera. A continuación se mostrarán los mejores resultados obtenidos en la fase de experimentación y los que se consideran más relevantes a efectos de cumplir los objetivos marcados.

El mejor resultado obtenido para el *fitness* del mejor individuo se puede observar en la Figura 6.1. Analizando la gráfica de los resultados se puede ver una mejora de la media de la población, aunque todavía no es demasiado significativa. Aún así, en este experimento se obtuvo el individuo con mejor *fitness* de los 30 experimentos realizados. Los datos obtenidos en este experimento se pueden observar en el Cuadro 6.1.

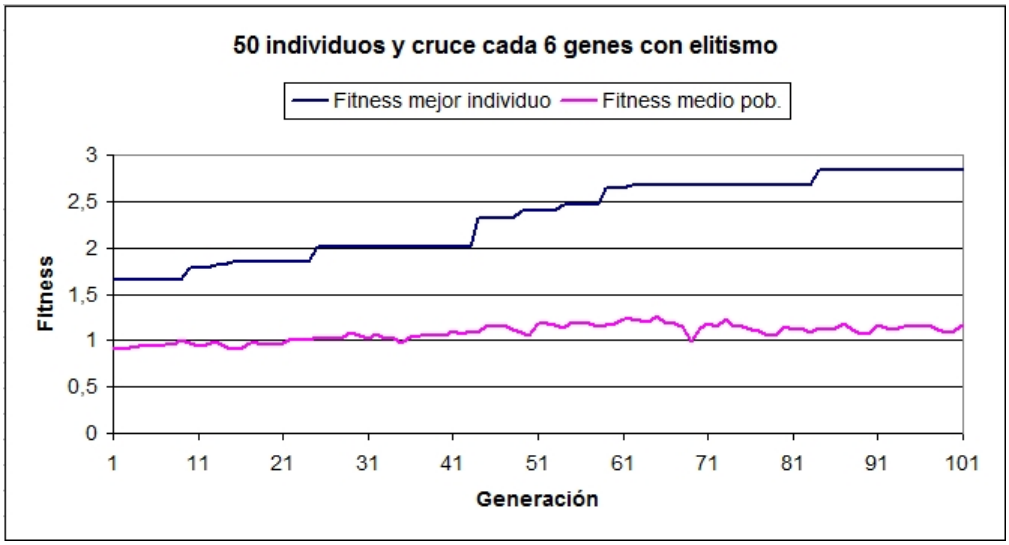

Figura 6.1: Experimento del individuo con mejor fitness

Tomando como referencia la progresión del *fitness* del mejor individuo se puede deducir que probablemente con más generaciones se podría obtener un individuo que mejorara el mejor de este experimento. Por lo tanto, teniendo en cuenta este dato y la tendencia obtenida en otros experimentos, alternativamente a los ya realizados, se realizarán experimentos en el que no se limiten por el número de generaciones totales, sino por el número de generaciones sin evolucionar del mejor individuo.

Mejor individuo	Media población	% Aumento fitness	Tiempo por generación
2,832452540	1,152638847	210.43 %	00:16:59

Cuadro 6.1: Resultados del mejor individuo

Con respecto a la media del *fitness* de la población, en la Figura 6.2 se pueden observar las gráficas obtenidas para el experimento en el cual se obtuvo la mejor media del *fitness* de la población. No obstante, comparando la gráfica de la Figura 6.2 de la media con la del experimento de la Figura 6.3 (y contrastándolo con los datos del Cuadro 6.2), se puede afirmar que, aún siendo ligeramente inferior la media, se ha conseguido obtener un mayor aumento de esta media respecto al valor de la media del *fitness* de la primera generación.

Con los datos que se han obtenido de estos 3 experimentos, seleccionados de los 30, y que se

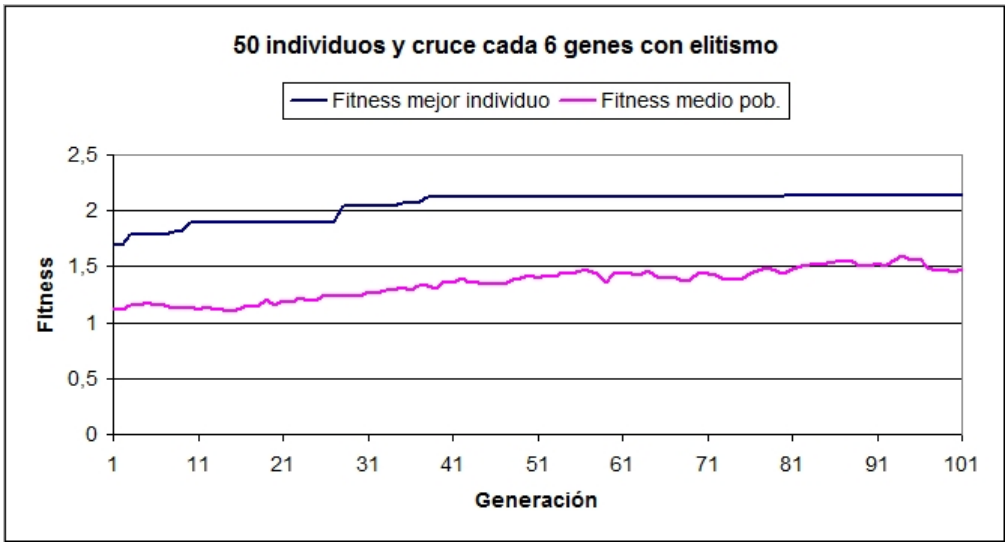

Figura 6.2: Experimento de la mejor media del fitness de la población

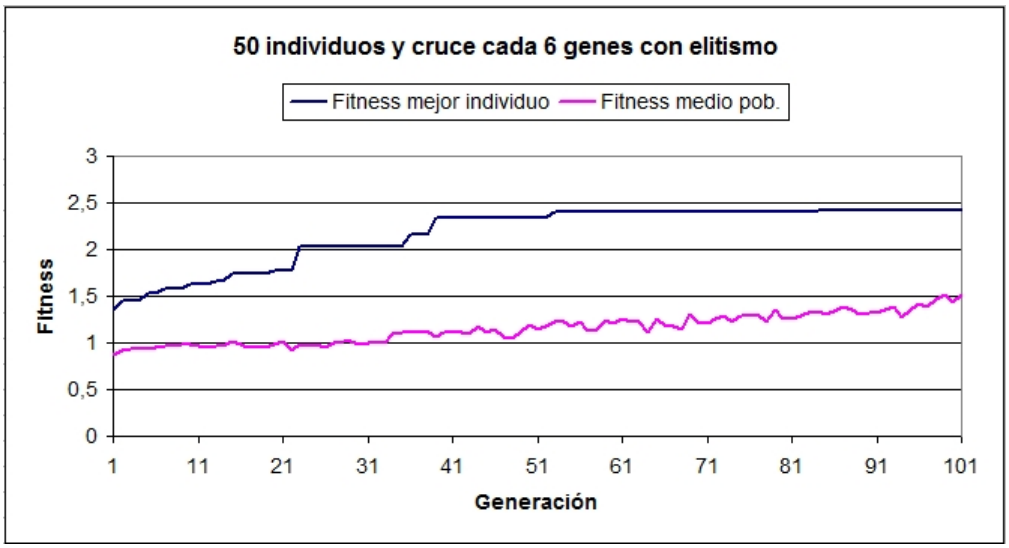

Figura 6.3: Experimento del mayor aumento de la media del fitness de la población

	Mejor individuo	Media población	%Aumento fitness	Tiempo por generación
Fig. 6.2	2.136969911	1.593677427	134.20 %	00:17:21
Fig. 6.3	2.412041559	1.519999412	164.35 %	00:17:09

Cuadro 6.2: Resultados de la mejor media y del mayor aumento de media

pueden ver en el Cuadro 6.3, se ha generado la gráfica de la Figura 6.4 en la cual se puede observar de manera visual la comparación entre los resultados obtenidos para los experimentos

comentados anteriormente.

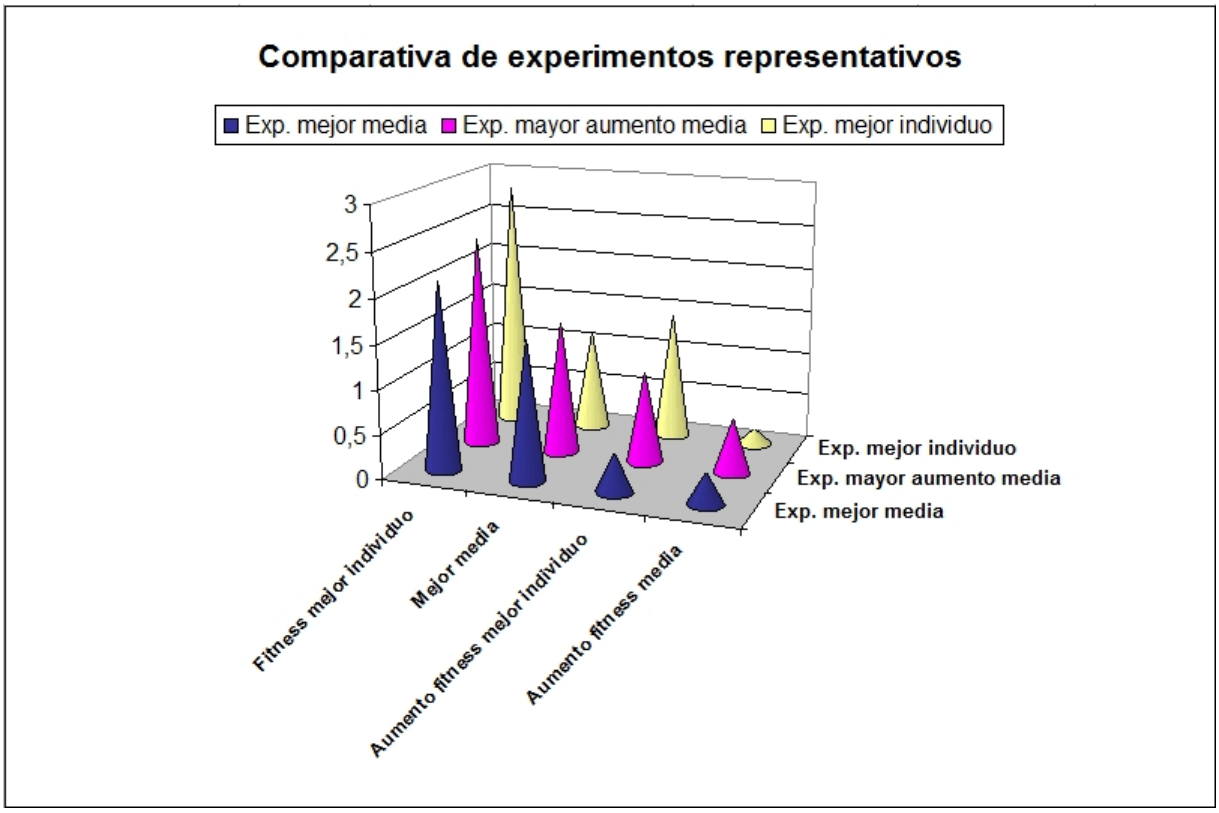

Figura 6.4: Comparativa de experimentos representativos

	Mejor individuo	Media población	%Aumento fitness	Tiempo por generación
Fig. 6.1	2,832452540	1,152638847	210.43 %	00:16:59
Fig. 6.2	2.136969911	1.593677427	134.20 %	00:17:21
Fig. 6.3	2.412041559	1.519999412	164.35 %	00:17:09

Cuadro 6.3: Cuadro comparativo de los experimentos representativos

Adicionalmente a los experimentes realizados con el cruce multipunto cada 6 genes, se han realizado 5 experimentos con el cruce multipunto cada gen y otros 5 con el cruce multipunto cada 62 genes. Con los resultados obtenidos de dichos experimentos se han obtenido los mejores individuos de cada tipo de cruce y el promedio de los datos censados en cada experimento.

Posteriormente, con dichos datos, se han realizado dos gráficas comparativas. La primera de ellas, Figura 6.5, muestra la comparativa de los promedios de los datos obtenidos en todos los experimentos. La segunda, Figura 6.6, refleja la comparativa de los mejores resultados obtenidos para cada uno de los cruces en los experimentos que se han realizado.

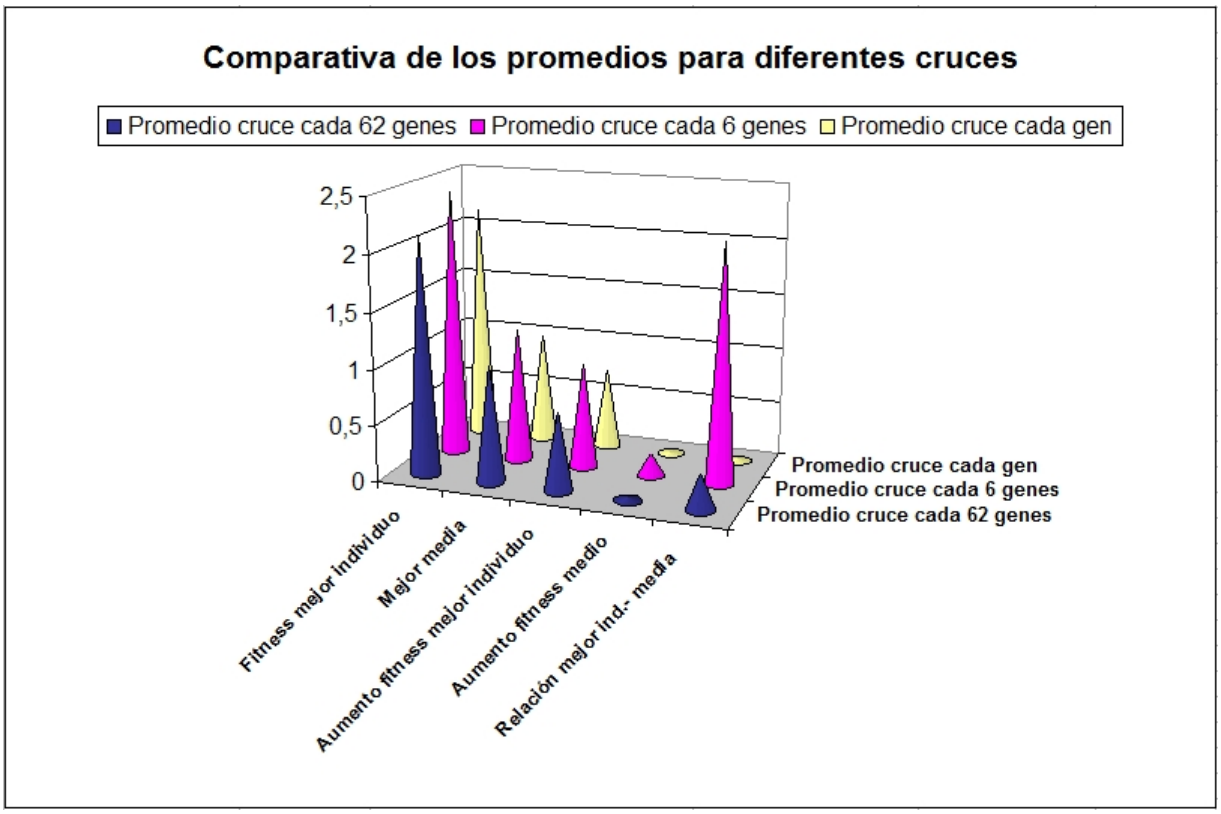

Figura 6.5: Comparativa de los promedios para diferentes cruces

Analizando estos datos, se puede ver que para todos los datos recabados, el tipo de cruce que se seleccionó para realizar la mayor parte de los experimentos, es decir, el cruce multipunto cada 6 genes, ha obtenido los mejores resultados. Es interesante la diferencia que se puede observar entre los diferentes tipos de cruce en la relación entre el *fitness* del mejor individuo de cada generación y el *fitness* medio de la misma. Esta relación es mucho mayor en el cruce cada 6 genes que los otros dos tipos de cruce.

Para el caso de la gráfica comparativa de los mejores resultados para los diferentes cruces se obtienen los mismos resultados que en el caso de los promedios, pero siendo más acentuada aún la diferencia en la relación entre el *fitness* del mejor individuo de cada generación y el *fitness* medio de la misma.

Los datos con los que han sido realizadas estas gráficas se pueden contrastar en el Cuadro 6.4 y en el Cuadro 6.5.

Analizando detenidamente las gráficas de los *fitness* de los mejores individuos para cada generación de todos los experimentos, y como se comentó anteriormente, se puede observar que la tendencia de crecimiento de la misma es positiva y que es bastante probable que siga aumentando. Esto es debido a que el número de generaciones que estuvo sin evolucionar la gráfica es

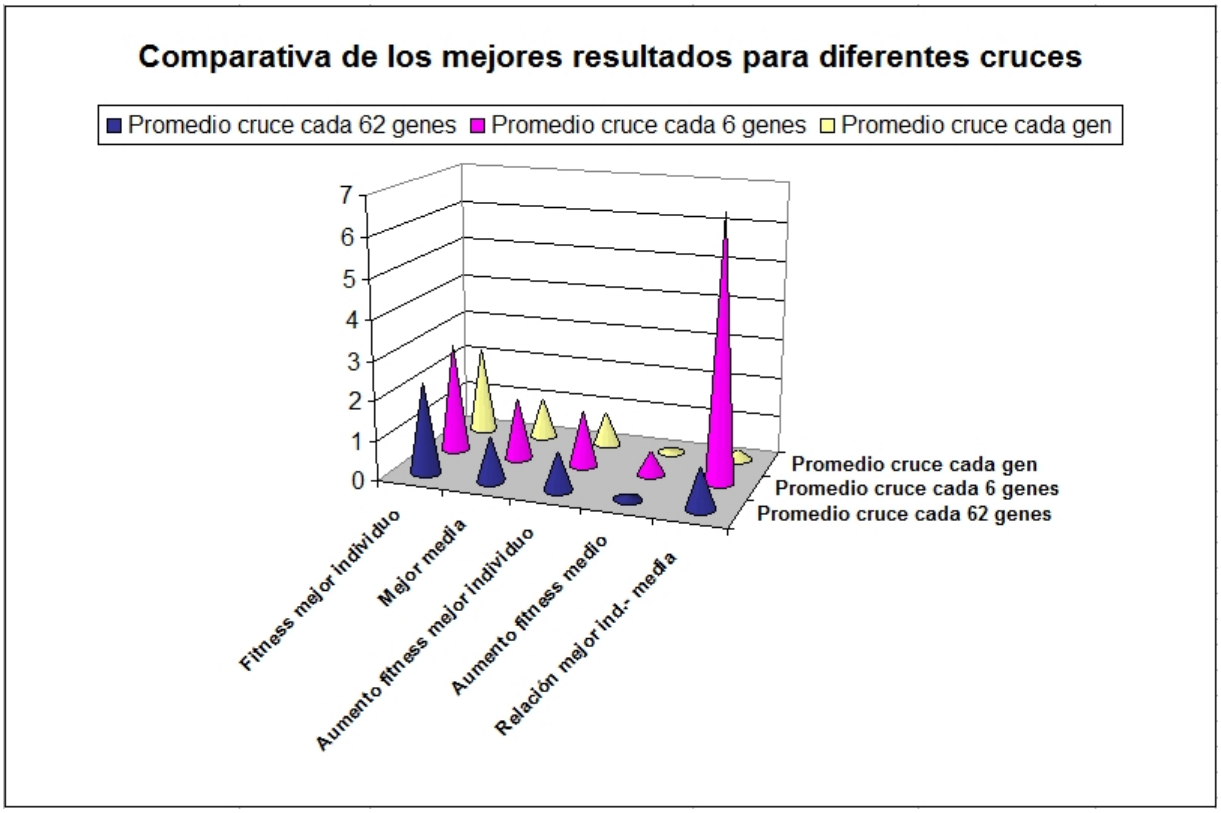

Figura 6.6: Comparativa de los mejores resultados para diferentes cruces

Cruce	Mejor individuo	Mejor media	Aumento mejor ind.	Aumento media pob.	Relación
1 gen	2.138590174	1.0652387	0.719771024	0.03928255	0.32568701
6 genes	2.413850222	1.2158409	0.962887607	0.21271817	2.14155511
62 genes	2.146058624	1.0027941	0.729054991	0,00507128	0.03221909

Cuadro 6.4: Cuadro comparativo de los promedios para diferentes cruces

Cruce	Mejor individuo	Mejor media	Aumento mejor ind.	Aumento media pob.	Relación
1 gen	2.344300352	1.17452496	1.006616289	0.1282147	1.07184492
6 genes	2.83245254	1.59367742	1.460397387	0.6321713	6.66463268
62 genes	2.23983519	1.04744667	0.876413193	0.0340927	0.26614043

Cuadro 6.5: Cuadro comparativo de los mejores resultados para diferentes cruces

mayor que el número de generaciones desde ese último avance y el final de la ejecución de los experimentos. Por este motivo se ha realizado un experimento sin limitación por generaciones, el cual realiza el algoritmo genético hasta que se realizan 500 generaciones consecutivas sin que mejore el *fitness* del mejor individuo. Las gráficas para este experimento se pueden observar en

la Figura 6.7 y los datos obtenidos en el Cuadro 6.6. Se ha conseguido una mejora del *fitness* del mejor individuo evolucionado en los experimentos anteriores de un 22.97 % y mostrando un *fitness* medio de la población ligeramente superior al de estos experimentos. Asimismo, se puede observar que la gráfica del *fitness* del mejor individuo se asemeja a una función logarítmica.

Los datos capturados para cada uno de los 30 experimentos realizados con el cruce multipunto cada 6 genes se pueden observar en el Cuadro 6.7, análogamente los datos de los experimentos para el cruce cada gen y cada 62 genes se pueden ver en los Cuadros 6.8 y 6.9 respectivamente.

	Mejor individuo	Media población	%Aumento fitness	Tiempo por generación
Fig. 6.7	3.480255461	1.716359035	281.00 %	00:16:20

Cuadro 6.6: Resultados del mejor individuo encontrado

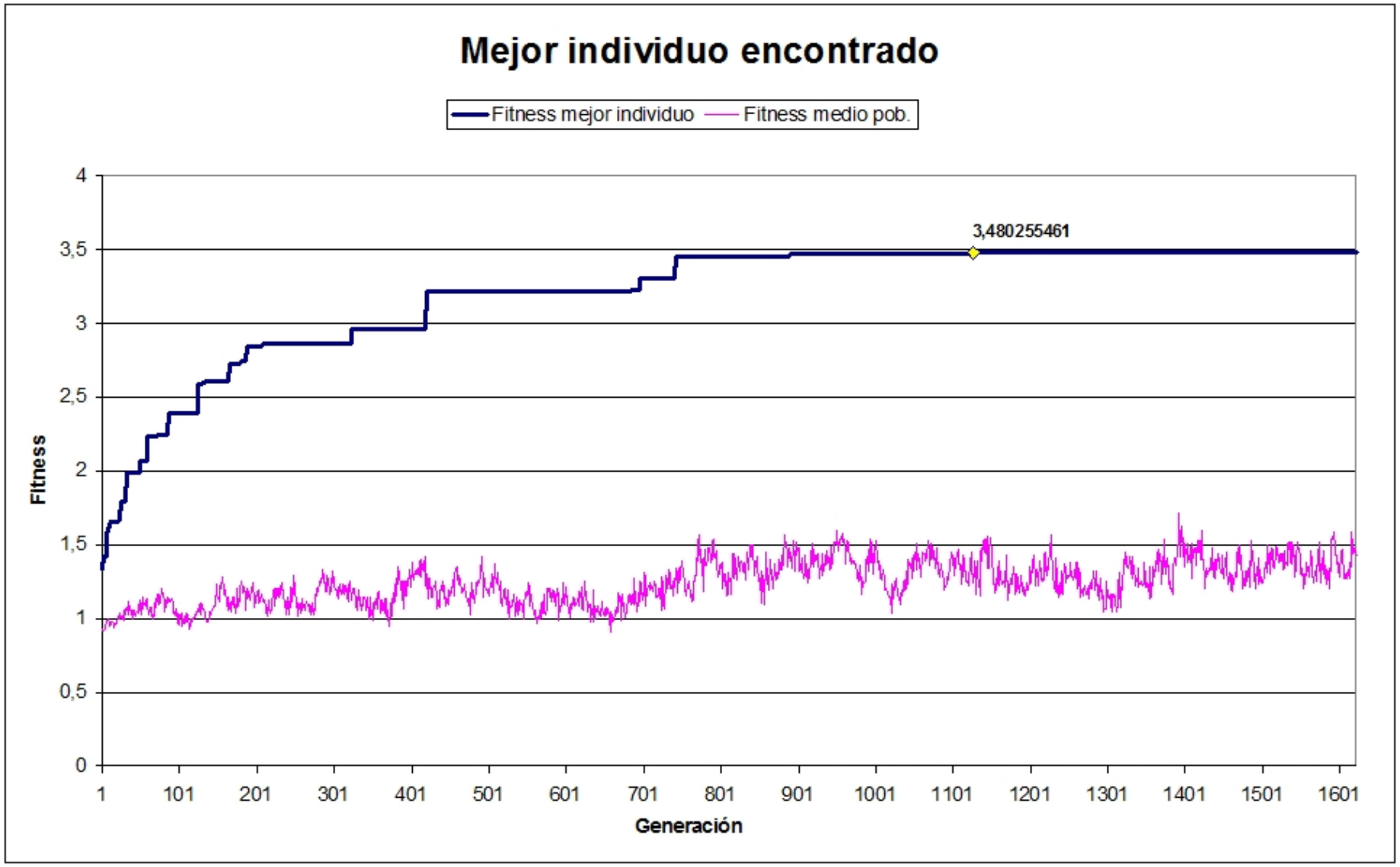

Figura 6.7: Mejor individuo encontrado

Mejor individuo	Mejor media	Aumento mejor ind.	Aumento media pob.	Tiempo por generación	Relación	% Aumento fitness
2.122131386	1.405030549	0.460862157	0.170577599	00:17:07	0.7861276	132.58 %
2.136969911	**1.593677427**	0.437042746	0.348686106	00:17:21	1.523907336	134.20 %
2.340031364	1.101001098	0.928629319	0.113687918	00:17:08	1.055739337	156.46 %
2.364067922	1.161783315	0.954952248	0.193580322	00:17:24	1.84859964	159.09 %
2.503095599	1.144450953	1.096461984	0.179762435	00:16:31	1.971026765	174.33 %
2.540241973	1.337947231	1.03579102	0.400552779	**00:16:27**	4.148889712	178.40 %
2.781227068	1.42902839	1.373289044	0.379827986	00:16:29	5.216136121	204.81 %
2.257180243	1.266924868	0.766260994	0.369267025	00:16:33	2.829549172	147.38 %
2.122476501	1.074822963	0.763799987	0.076744371	00:16:53	0.586173494	132.62 %
2.425106495	1.122971646	1.062624723	0.117107932	00:17:16	1.244417835	165.78 %
2.265030606	1.211118818	0.756082361	0.173855548	00:17:18	1.314491133	148.24 %
2.350285212	1.195232794	0.901355313	0.204524431	00:17:19	1.84349183	157.58 %
2.237484114	1.06787777	0.647437724	0.114258067	00:17:00	0.739749831	145.22 %
2.770627796	1.187532968	1.447486325	0.212477551	00:16:50	3.075583499	203.65 %
2.24451734	1.078618366	0.896132643	0.000341964	00:17:28	0.003064454	145.99 %
2.230240677	1.128166458	0.862534911	0.090556931	00:17:29	0.781085146	144.43 %
2.776996175	1.211906254	1.318434016	0.186098052	00:17:22	2.45358002	204.35 %
1.969927085	1.115281321	0.648586664	0.084971846	00:17:00	0.551116062	115.90 %
2.831019817	1.259000858	1.181891474	0.241973319	00:16:34	2.859862027	210.27 %
2.548783364	1.183958297	1.052743288	0.148025219	00:16:41	1.558325562	179.34 %
2.584012526	1.179095828	1.086960959	0.244955565	00:16:45	2.662571359	183.20 %
2.20709113	1.151188688	0.656517061	0.224523016	00:17:19	1.474031907	141.89 %
2.238273904	1.081964039	0.872928106	0.123490934	00:16:58	1.077987069	145.31 %
2.41530158	1.154067732	0.812708951	0.079479587	00:17:29	0.645937714	164.71 %
2.83245254	1.152638848	**1.460397387**	0.188176604	00:16:59	2.748126203	**210.43 %**
2.764460691	1.352340362	1.365112175	0.439848775	00:16:54	6.004429186	202.98 %
2.316731629	1.175917937	1.059584674	0.216585841	00:16:37	2.294910373	153.91 %
2.412041559	1.519999412	1.05424474	**0.6321713**	00:17:09	**6.664632684**	164.35 %

Cuadro 6.7: Cuadro de los datos de los 30 experimentos para el cruce cada 6 genes

Mejor individuo	Mejor media	Aumento mejor ind.	Aumento media pob.	Tiempo por generación	Relación	% Aumento fitness
1.904162502	0.974441843	0.273676588	-0.022407863	00:17:12	-0.061325076	108.69 %
2.123906301	1.073764188	0.816028459	0.031759796	00:16:54	0.259168971	132.77 %
1.979371288	1.072461843	0.666557405	0.06869632	00:16:51	**0.457900405**	116.93 %
2.344300352	**1.174524969**	**0.835976379**	0.128214738	**00:16:44**	1.071844927	**156.93 %**
2.341210426	1.031000874	1.006616289	-0.009850242	00:16:46	-0.099154143	156.59 %

Cuadro 6.8: Cuadro de los datos de los 5 experimentos para el cruce cada gen

Mejor individuo	Mejor media	Aumento mejor ind.	Aumento media pob.	Tiempo por generación	Relación	% Aumento fitness
2.16856618	1.00390065	0.780636805	**0.034092734**	00:17:06	**0.26614043**	137.67 %
2.23983519	0.981555626	**0.876413193**	-0.018516592	00:17:15	-0.162281859	**145.48 %**
2.218106146	**1.04744667**	0.720415045	0.017782263	00:17:12	0.1281061	143.10 %
1.951928897	0.973628004	0.571482112	0.012134309	00:17:27	0.069345404	113.92 %
2.151856705	1.007440012	0.6963278	-0.02013629	**00:16:52**	-0.140214587	135.84 %

Cuadro 6.9: Cuadro de los datos de los 5 experimentos para el cruce cada 62 genes

Capítulo 7

Conclusiones

En este proyecto fin de carrera se recoge el estudio y la optimización del patrón de movimiento del robot Aibo de Sony para que este recorra la mayor distancia posible. Con este objetivo se ha analizado el patrón que ofrecía el controlador URBI, tanto en el simulador como en el robot físico, y se ha desarrollado una metodología para permitir su aplicación práctica. Para ello, se han utilizado técnicas de computación evolutiva con ensayos experimentales en un simulador Webots. Como resultado se ha obtenido un patrón de movimiento que mejora ampliamente el patrón tomado como individuo base en los experimentos.

Por medio de la aplicación del algoritmo genético desarrollado se ha conseguido mejorar la cantidad de espacio recorrido por el robot en un número de pasos determinados, aumentando en un 281 %, la distancia euclídea recorrida por el individuo base aportado por el controlador URBI para el robot.

Cabe destacar el buen comportamiento general del algoritmo evolutivo en un espacio de búsqueda del tamaño del especificado en los experimentos que se han llevado a cabo. Este espacio de búsqueda está compuesto por 372 genes representados por números reales que pueden tomar cualquier valor dentro de los rangos de movimientos de las articulaciones del robot.

También se ha conseguido mejorar el *fitness* medio de la población en comparación con el de la población aleatoria inicial. Si bien este aumento no se ha mantenido estable a lo largo de cada experimento ni ha sufrido un aumento constante, es lo suficientemente significativo para ser valorado como positivo. Esto es debido a que, como se ha mencionado anteriormente, existe un espacio de búsqueda enorme y a que cualquier pequeña variación en uno de los genes del individuo puede causar un gran cambio en el patrón de movimiento. Cada movimiento de una articulación del robot afecta al siguiente movimiento, por lo tanto, estas pequeñas variaciones pueden derivar en grandes diferencias en el *fitness* del individuo. A esto hay que añadirle el ruido producido por las imprecisiones de los motores del robot.

Por este mismo motivo, la aplicación del elitismo con reemplazamiento en el algoritmo genético ha ayudado a la mejora del mismo, ayudando a conseguir evolucionar los individuos y a obtener los resultados expuestos anteriormente. La motivación y la justificación de su uso se debe a que, para varias ejecuciones distintas del patrón de movimiento de un individuo, se pueden obtener distancias euclídeas diferentes como resultado de las leyes físicas del entorno y de la variación en el estado del robot que genera un movimiento para los movimientos posteriores. Este punto es una de las mayores dificultades que se han encontrado, puesto que es posible que se hayan descartado individuos que podían haber evolucionado más rápidamente o mejor.

Todas las herramientas de desarrollo utilizadas, así como el simulador y el controlador para el robot son de libre distribución y versiones gratuitas o de evaluación. Esto ha supuesto algunos inconvenientes en el desarrollo y realización del presente proyecto fin de carrera. Se han producido cambios de versiones en el controlador URBI, lo cual ha conllevado el cambio del código fuente de las funciones realizadas para el control del robot. Además, al utilizarse una versión de evaluación del simulador Webots se ha encontrado la limitación temporal del uso del mismo para las funciones de comunicación con el controlador URBI, por lo tanto se han tenido que implementar diferentes métodos y planificaciones para conseguir evitar esta limitación en el uso del simulador. Dentro de los cambios necesarios para obtener una funcionalidad suficiente para desarrollar los experimentos, se ha tenido que cambiar la fecha del servidor y reiniciar el entorno del simulador para evitar la limitación temporal de su uso. Como contrapartida se ha penalizado cada generación del algoritmo genético con 2 segundos de espera por cada 10 individuos. Este tiempo es el necesario para reiniciar el entorno del simulador y evitar la limitación temporal de uso.

Por último, destacar la dificultad de realizar proyectos de investigación con herramientas gratuitas y sin contar con presupuesto, ya que limita la funcionalidad y la posibilidad de realizar el mismo en un tiempo menor. Por otro lado, este hecho, que a priori puede suponer un inconveniente, se convierte en un aliciente de superación y de motivación para encontrar soluciones más eficientes. Así que ésta dificultad ha servido para adquirir un mayor conocimiento de herramientas libres, y ha permitido aumentar la capacidad de solucionar problemas reduciendo el coste, aspecto siempre deseable en cualquier organización y proyecto.

7.1. Líneas futuras de investigación

El trabajo realizado en el presente proyecto fin de carrera ha llevado al desarrollo de un patrón de movimiento para el robot Aibo. Sin embargo, este trabajo no puede considerarse como un final de una línea de investigación, sino como un inicio para una línea de trabajo que permita profundizar en la aplicación de los algoritmos evolutivos a distintos problemas reales de ingeniería, y más concretamente de la robótica.

Este último punto debe tenerse muy presente ya que este proyecto no se ha desarrollado como

una aproximación teórica, sino como una búsqueda de soluciones a problemas reales y una optimización de soluciones previas encontradas por otros métodos. En esta búsqueda de aplicación a casos reales se han ido detectando deficiencias y necesidades adicionales de la metodología del diseño inicial. Es por ello que el trabajo futuro que se plantea no implica únicamente mejoras en los procesos de búsqueda y simulación, sino facilitar su aplicación en nuevos problemas de índole similar.

Esto supone que las futuras líneas de investigación puedan resumirse en los siguientes puntos:

- La aplicación de los resultados en un robot Aibo real una vez obtenido un controlador para el robot en el entorno del simulador. Se podría cargar en el robot para realizar una comparación de los resultados obtenidos en el simulador y en el robot real.
- Las necesidades específicas de otros problemas pueden hacer que existan algoritmos de búsqueda más eficientes para cada caso. La creación de una librería formada por un grupo de diferentes herramientas para cada fase del algoritmo genético puede facilitar el evolución de los experimentos y la resolución de los mismos, mejorando la exploración del espacio de búsqueda.
- La posibilidad de utilizar diferentes funciones de evaluación puede hacer que se realice un mayor refinamiento en los objetivos planteados en el presente proyecto y que pueden servir para una ampliación del mismo para próximos estudios. Entre estas modificaciones se encuentra la inclusión de la desviación angular del robot para conseguir que recorra la mayor distancia posible en linea recta.
- Otra linea de investigación viable es la aplicación de otras técnicas de inteligencia artificial en colaboración con los algoritmos genéticos, como podría ser la utilización de agentes autónomos cooperativos para cada una de las extremidades del robot, realizando un controlador por medio de algoritmos genéticos para cada uno de ellos. De esta manera se conseguiría un patrón de movimiento más versátil en caso de cambios en el entorno que rodea al robot.
- Como última línea se propone la inclusión de los acelerómetros, giroscopios y sensores de proximidad del robot tanto en la función de evaluación de los individuos del algoritmo genético como en el supuesto de utilizar agentes autónomos utilizarlo como entrada de estímulos al agente para que este reaccione.

Bibliografía

[1] M. Anthony Lewis, Andrew H. Fagg, and Alan Solidum. Genetic programming approach to the construction of a neural network for control of a walking robot. In *IEEE International Conference on Robotics and Automation*, pages 2618–2623. IEEE Press, 1992.

[2] G. S. Hornby, M. Fujita, and S. Takamura. Autonomous evolution of gaits with the sony quadruped robot. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1297–1304. Morgan Kaufmann, 1999.

[3] G.S. Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto, and M. Fujita. Evolving robust gaits with aibo, 2000.

[4] Roberto Guzmán y Rocío Alaiz M^{a} Concepción Marcos. Autoguiado de robots móviles mediante redes de neuronales. In *XXV Jornadas de Automática*, 2004.

[5] Ricardo A. Téllez y Cecilio Angulo. Generando un agente robótico autónomo a partir de la evolución de sub-agentes simples cooperativos. In *W WORKSHOP EN AGENTES FISICOS [84-933619-6-8]*, pages 113–118. Angulo, 2004.

[6] Gregory S. Hornby, Seichi Takamura, Takashi Yamamoto, and Masahiro Fujita. Autonomous evolution of dynamic gaits with two quadruped robots. *IEEE Transactions on Robotics*, 21:402–410, 2005.

[7] Sony Corporation. Open-r sdk model: Information for ers-7. http://www.robots.newcastle.edu.au/QuinlanThesis/ModelInformation_7_E.pdf, 2004.

[8] François Serra and Jean-Christophe Baillie. Aibo programming using open-r sdk tutorial. http://www.ensta.fr/~baillie/tutorial_OPENR_ENSTA-1.0.pdf, 2003.

[9] Sony Corporation. Open-r sdk: Level 2 reference guide. http://paginas.fe.up.pt/~eol/ROBO/20022003/robotica/documents/Sony/ /Level2ReferenceGuide_E.pdf, 2002.

[10] Ricardo A. Téllez. Manual de r-code v2.1. http://www.ouroboros.org/rcode2v1.pdf, 2005.

[11] Urbi doc for aibo ers2xx ers7 and urbi 1.0. http://www.gostai.com/doc/en/aibo.

[12] Urbi tutorial for urbi 1.0. http://www.gostai.com/doc/en/urbi-tutorial-1.0.pdf, 2006.

[13] Página web de urbi for aibo. http://www.gostai.com/aibo.html.