Distributed neural control for complex autonomous robots

Ricardo A. Téllez

Submitted in total fulfilment of the requirements
of the degree of Doctor of Philosophy

February 2010

Artificial Intelligence
Technical University of Catalonia

Thesis advisor: Cecilio Angulo

To Mayte and to the memory of our best friend, Woody.

The opposite of a modular system is a coupled one.

In a coupled system, changes in one part trigger changes in another.

Coupling leads to complexity.

Complexity leads to confusion.

Confusion leads to suffering

This is the path to the Dark Side.

Paul Fitzpatrick
Towards long-lived robot software,
Workshop on Humanoid Technologies,
Humanoids 2006

Abstract

This thesis is about generating neural network based controllers for complex robots. For such goal, one of the main problems is how to obtain a trained neural network which successfully controls the robot. A promising methodology is the use of evolutionary robotics (ER). However, up to date, ER methods do not achieve the generation of behaviors for complex robots with a fixed body structure composed of lots of sensors and actuators. No satisfactory results exist due to the large search space that the ER algorithm has to face. Furthermore, the *boostrap* problem prevents the generation of simple solutions with a minimum fitness value that could guide the evolutionary path towards the final solution. Solutions like incremental evolution try to overcome the problem, but they do not scale well in complex robots with lots of devices.

The question is then, why natural evolution succeded evolving complex animals, but ER do not. One answer to that question may be that, while natural evolution gradually evolved at the same time the animals body plan, their sensors and actuators, their nervous system, and even their environment, artificial evolution tries to evolve the nervous system for a robot with a fixed given body, sensors and actuators, within a fixed complex environment.

Our proposal states that when none of the evolutionary constraints can be relaxed, then it will be mandatory the use of external knowledge to guide the evolutionary process. ER approaches try to avoid the use of such knowledge (called bias) because it directs the evolutionary search towards specific solutions. In this work, we advocate instead for the use of bias as an inevitable situation when the robot body, task and environment are **complex** and **fixed**.

Based on this idea, we develop a modular architecture for evolutionary controllers called DAIR, which allows the **selective** introduction of bias in the evolutionary controller. The architecture allows the introduction of external knowledge on **selected** stages of the evolutionary process, affecting only **selected** parts of the controller that need to accomodate that information. The evolutionary controller is *progressively designed* in a series of stages, almost in a quirurgical way, independently of the complexity of the robot (in terms of number of sensors and actuators). This approach allows to avoid the boostrap problem completely, and to obtain a completely distributed controller for the robot using only artificial evolution.

We will show how to apply our method in general robots, and then we will apply it to a complex Aibo robot in tasks like walking, standing up, or learning to touch the ground, both in simulation and real robot. Additionally, we will show how a DAIR controller can be influenced from external systems by using a tonic signal. On a final stage, we will study how the architecture elements build an internal representation of the outside world based on its experiences during the evolutionary process.

Keywords

Neural networks, complex robots control, evolutionary robotics, modularity, inner world

De bien nacido es ser agradecido

(popular Spanish proverb)

The creation of such a work like a thesis takes a long time and hard work. During that time, several people helped me to achieve that goal. Here is my tribute to all of them (I hope I did not miss anybody!).

I will start thanking my thesis advisor Dr. Cecilio Angulo. His big help, support and good advices made possible that the thesis had a start and an end. I would like to thank also all the people in the GREC research group, specially Dr. Andreu Català, for providing me such good working environment and making of me part of the group. Thanks to Mercè Cabané, Xavier Parra, Pedro Ponsa, Cristóbal Raya and Francisco Javier Ruiz. Special thanks go to Diego Pardo with who I spent long research hours.

My thesis tutor Jordi Delgado and my two thesis teachers Ton Sales and Mario Martin provided me with useful advices at the beginning of the thesis. Thank you for all those.

Professor Alicia Casals provided me with insightful suggestions and pointed me to the GREC research group. She also allowed me to be in the Euron board, being this a beautiful experience. Thank you.

I would like to thank all the people at the Agents Research Lab of the University of Girona, specially to professor Beatriz López and PhD student Christian Quintero who allowed me to start my research with the Aibo robot. Many thanks to professor Josep Lluis de la Rosa and all the students I met there: Bianca Inocentti, Silvana Aciar and Salvador Ibarra.

My stay at the BIRG group at EPFL was very pleasant thanks to all the people there, and specially to professor Auke Ijspeert. Thanks to the other PhD students I met there with who I spent both research and leissure moments: Luca Righetti, Jonas Buchli, Yvan Bourquin, Alessandro Crespi, Joel Rossier, Marlise Taric, André Badertscher, Pierre-André Mudry, Adamo Maddalena, Raphaël Haberer-Prous and Rafael Arredondo.

Many thanks go to my friend Olivier Michel, CEO of Cyberbotics, and his beatiful wife Valérie. They made my stay in Laussane like in home.

Thanks also go to professor Jun Nishi for providing the opportunity to explain my research at University of Taniguchi in Japan.

I would like also to thank to other people who also influenced me during different periods of the thesis. Those are my friends Maximo Pedraza, Sergio Moreno and Oscar Vilarroya.

Special thanks go to my teacher and friend Josep Maria Torrents. He is been always there with a good advice. In fact, he indirectly influenced me to start the thesis. Thank you for that.

Special thanks go to my fathers Ricardo and Nieves, and to my brothers Benjamin, Nieves and Nuño. I am who I am now thanks to their influence.

The most special thanks go to my best friends in the world, Mayte, Woody, Lu, Rasputin and Lola, the only ones who unconditionally supported me from the very beginning. This was possible to all of you. Muchísimas gracias.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research

Ricardo A. Téllez

Barcelona

February, 2010

Contents

1 Overview

The opening chapter of this thesis consists of a concise overview of the entire thesis. It succinctly describes the main area of research, the methodology applied, and the results obtained. It also includes a summary of contributions[1].

1.1 General overview

This thesis focuses on the creation of behaviours in *complex robots* using artificial neural networks (ANNs) as information processing elements in a distributed neurocontrol architecture. The term "complex robot" is understood as a physical agent composed of a large number of sensors and actuators. The generation of neural controllers for these robots is a complex task due to network training. In most cases training sets for the task at hand are not available, and even when examples are available it is still unclear as to precisely how to allocate error to the different components involved in the control of the robot for the given task.

One possible approach is the use of evolutionary robotics methodology for network training. Evolutionary robotics (ER) uses evolutive algorithms for network weights updating [Nolfi and Floreano, 2000], avoiding training examples and blame assignment. The use of ER however, presents its own set of problems; it requires real interaction between the robot and its environment, and it does not scale well as the complexity of the robot and/or its behaviour grows up. This thesis will deal with these problems by developing a general distributed control architecture for robots. This architecture is based on neural networks as basic processing elements, and will be independent of the task, the environment, and the robot. The architecture is therefore proposed as a solution to the problem of generating behaviours in complex robots within the evolutionary robotics paradigm.

[1] For an extended overview, including videos, papers and additional images and results, visit: http://www.ouroboros.org/thesis

1.1.1 AI robotics

One of the main issues when developing AI systems is the artificial creation of an autonomous *entity*[2] that behaves *intelligently* in a real life environment. Several examples exist, from providing care giving assistance to the frail or handicapped, to the completion of highly complex tasks in extremely dangerous environments, and ranging right through from dazzling entertainment to mundane, tedious and repetitive tasks. To design such physically situated autonomous agents, two integral branches of science and engineering are required to work together: robotics and artificial intelligence. The combination of these two disciplines gives birth to what is referred to as *AI Robotics* [Murphy, 1998], that is, the creation of physical autonomous robots that have their behaviour controlled by artificial intelligence-based algorithms. Whilst robotics provides the physical basis which the *entity* will rely on, artificial intelligence provides the control mechanisms for the intelligent use of that body structure in order to complete tasks in the real world.

We will refer to the software control program as *the controller*, designed to perform the robot's behaviour, since it *controls* the physical robot body based on the information the sensors and actuators provide it with. A controller must have an *architecture* in order to have a principled way of organizing the control system [Mataric, 1992]. Control architectures can be defined as [Russell and Norvig, 2003]

> the practical structure of a robot's software which defines how the job of generating actions from percepts is organized

Or, otherwise stated [Bekey, 2005]

> the software which defines the way in which sensing, reasoning, and actions are represented, organized, and interconnected

The research introduced in this thesis is situated within the AI robotics framework. The thesis is however exclusively dedicated to the subject of artificial intelligence, produced by the development of neural control and learning mechanisms. Even if emphasis is placed on the software development, the physical robot body plays an important role in the research by providing the substrate upon which the architecture relies. As will become apparent in due course, the proposed architecture will accommodate the interaction between the robotic body and its environment in order to accomplish a task[3], according to the principles stated by *new AI* [Pfeifer and Bongard, 2007], where intelligence cannot be understood without a body.

As a main requirement, the architecture to be developed in this thesis will be independent of what the physical robot body is composed of. The architecture

[2] Otherwise referred to as an agent.

[3] In this thesis the term *behaviour* will be used in a general manner along with the term *task*, indicating in both cases the same concept of robot purpose. Whilst the term *task* is more commonly used in the robotics field, the term *behaviour* is the preferred choice of the AI domain.

can therefore be applied to any type of robot, completely independent of the number and type of devices it contains.

The actual implementation of a robot controller within the AI paradigm can be performed using a variety of different frameworks, including; subsumption architectures, schema theory, dynamical systems, fuzzy logic, or Bayesian systems. This thesis uses ANNs as processing elements for the controllers. Neural networks are an interesting option for use as controllers for several reasons, and their capacity to generalize (i.e. to provide a similar response in similar situation) is certainly one of the most interesting. The main effort when using neural networks is the amount of training required to perform any given task, and this can be undertaken using one of the three main groups of training techniques; supervised learning [Kotsiantis, 2007], unsupervised learning [Ghahramani, 2004] and reinforcement learning (RL) [Sutton and Barto, 1998]. For this thesis, unsupervised learning methods have been discarded because of their focus on clustering data. Supervised learning has also been discarded, as network learning would require a set of examples for the network *behaviour*, and these examples are not normally available. RL networks, on the other hand, learn from a signal that indicates to them how well or how poorly they are performing the task. For the generation of our controllers, we will select a reinforcement training method known as *evolutionary robotics*[4] (ER) paradigm [Nolfi and Floreano, 2000], due to the lack of training examples for most of the cases.

1.1.2 Evolutionary robotics

Evolutionary robotics is a research area within robotics that makes use of evolutionary algorithms such as genetic algorithms for the generation of robot controllers. Evolution-based techniques are particularly interesting to work with as they present a wide range of possible solutions for any given problem. The evolutionary algorithm performs a search through the range of possible solutions and establishes which solution is the most suitable for the problem at hand. This solution may at times appear counter-intuitive from an engineering perspective, but it liberates the engineer from the painstaking task of having to detail every last aspect of the system. When evolutionary algorithms are used to generate a robot controller, it is said that the controller has been *evolved*.

Evolutionary robotics is a good method for the training (i.e., evolution) of artificial neural networks. The evolutionary algorithm can find the required weights and structure for the ANN to perform the desired computation without requiring a set of training examples, which are not usually available in robot control. Despite its fascinating advances, current evolutionary robotics is still not powerful enough to evolve controllers for robots composed of a large set of actuators and sensors, and the current field of evolutionary robotics is mainly relegated to the domain of wheeled robots performing simple tasks. When the robot to be controlled is complex in terms of number of sensors and actuators,

[4] We consider evolutionary robotics a RL method, even if this fact is not accepted by everyone. For a discussion on whether ER is or is not a RL method, see [Sutton and Barto, 1998], page 9.

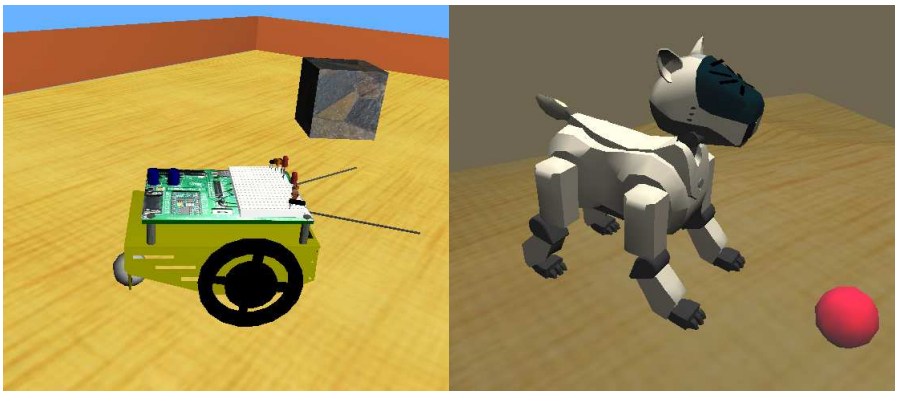


Figure 1.1: A simple robot (left, the Boebot) and a complex one (right, the Aibo robot). A simple behavior to move around those robots, like for example, a wall following behavior, implies, in the case of the Boebot robot, the evolution of a controller for 4 devices (sensors and actuators). The same behavior for the Aibo robot implies the evolution of a controller able to handle between 25 and 30 devices (depending on implementation).

the search space for the evolutionary algorithm becomes immense, and the algorithm is unable to find a solution within a reasonable amount of time. Moreover, when the controller to be evolved is complex in terms of the task, the complexity of the search space may become so high that it could become impossible to find even an initial partial solution with which to guide the evolutionary process toward the final solution.

In this thesis, a solution is proposed to these types of problems. A distributed architecture for the control of complex robots using ANNs has been created. It makes use of evolutionary techniques to find the most suitable network weights. The distributability of the architecture makes it possible to partially overcome the curse of the dimensionality problem usually encountered in evolutionary processes. The architecture is of a general purpose in the sense that it is independent of the robot and the task to be solved. The architecture is of the *reactive* type [Braitenberg, 1984, Brooks and Stein, 1994]. Additionally, the architecture allows for the coupling with a *deliberative* process [Albus et al., 1987], what may end in a *hybrid* approach [Gatt, 1993, Murphy, 1998].

1.2 The thesis in a nutshell

The main goal driving this thesis is the generation of general controllers for complex robots like the quadruped 16 degrees of freedom (DOF) Aibo robot (figure 1.1, right). These robots are composed of multiple sensors and actuators that all need to be perfectly coordinated in order to perform even the simplest of tasks, such as moving around. This research interest is to implement a method

for the generation of neural network-based controllers for these types of robots, and evolutionary robotics is a good framework to work with. Unfortunately, as will be described in chapter 2, for these robots the evolutionary search space can be so large and complex that the evolutionary process is not able to find any solution unless a certain amount of external knowledge is introduced into the evolutionary process. The introduction of external knowledge is something that is usually avoided in evolutionary robotics as it constrains the capacity of the genetic algorithm to find novel solutions for the problem. It can, and will be argued however, that when robots and behaviours are given and complex, the use of external information in the evolutionary process is unavoidable.

As will be described in much finer detail in chapter 2, several paths can be followed in order to overcome the problem of using evolutionary robotics in complex robots performing complex tasks: to develop a better encoding method, to develop a better genetic algorithm, or to develop better control architectures, as well as any combination of these, to a certain extent. This research addresses the problem of developing a better control architecture with the following characteristics:

- Complex robot evolution: it must allow the evolution of controllers for complex robots, that is, robots composed of a high number of devices.
- Unspecified robot: it has to be independent of the type of robot used, and must be able to work for wheeled, legged and even static robots.
- Selective information use: the control architecture must allow the selective introduction of external information into the evolutionary process, and the introduction of this knowledge must affect only the specific parts of the controller involved in one particular part of the behaviour.

The capacity for the introduction of external knowledge is one of the key priorities for the architecture; when none of the evolutionary requirements (that is, the robot body, the robot task and the robot environment) can be relaxed, and they are complex, then the use of external knowledge becomes mandatory. It is for this purpose then that the use of selective information is one of the key requirements of the architecture. When a neural network is evolved, all the weights are evolved at the same time, and all of them are influenced by the evolution process. In the case of complex controllers however it could be preferred to affect only a part of the controller with the information, and the selective information requirement therefore assures that the information introduced will only very slightly affect the other parts of the controller.

Selective information requirement leads to the selection of a modular controller. A modular controller would have the ability to evolve certain particular modules whilst the others remain constant. Furthermore, the robot platform independency requirement also leads to the use of modular controllers as it becomes necessary to design a flexible and scalable enough architecture in terms of the number of sensors and actuators.

The modular approach is not a new approach in the field of neural control. Modular approaches have however always focused on the behavioural division

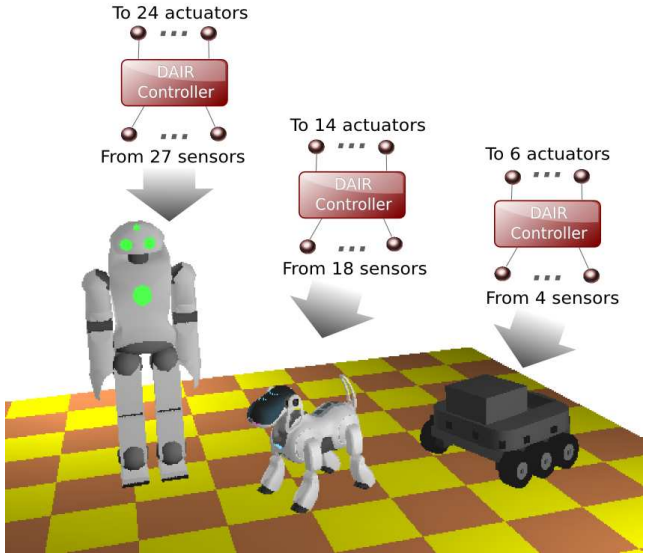


Figure 1.2: The DAIR architecture, described along the thesis, applies to any type of robot, independently of its number and type of devices.

of the task, i.e., the generation of modules at the behavioural level. This type of modularization, however, has yet to prove capable of performing the evolution of controllers for robots with large amounts of sensors and actuators. Chapter 3 therefore introduces a review of current available neural modular architectures and proposes an architecture which meets all three preceding requirements. A new type of modularization at the device level is proposed, where each device will be associated with its own independent processing module, whilst simultaneously maintaining an influence on the rest of modules. Chapter 4 describes the proposed modularized architecture, named DAIR (Distributed Architecture with Internal Representation), and explains how it can also easily be combined with a typical behaviour-based modularization [Brooks, 1991] in a single controller, thus allowing for a deeper level of modularity.

The resulting controller obtained from the DAIR architecture will in fact be a distributed one, since there is no central component coordinating the smaller modules. Instead, each module manages its own associated device (sensor or actuator), controls it, and at the same time keeps it coordinated with the rest of modules in order to ensure a globally coordinated action. During the evolutionary process, each module is trained to master its associated device, and along with this process, coordination with the rest of modules is also learned. Coordination between modules is possible due to the existence of connections between the modules, which are also evolved (see figure 1.2).

Due to the fine-grained modularity of the architecture, the evolution of modules and connections can be performed either in separate processes, or all at the same time. When either the task at hand, or the robot to evolve is simple, everything can be evolved at the same time, as will be demonstrated in section 4.2.6 and section 4.3. In both cases, the DAIR architecture has proven to perform better than most of the typical evolutionary architectures, as will be discussed in chapter 5.

The strongest performance of the architecture however, is obtained when a complex robot is used, and chapter 6 describes how the DAIR architecture overcomes the problem of evolving a complex behaviour in a complex robot. The architecture allows for a *modularization of learning* by implementing what we refer to as the *progressive design* of a controller. The distributability of the architecture allows for the selective evolution of the different parts of the controllers, resulting at the same time in a coherent behaviour between all of them. This fact completely avoids the bootstrap problem which prevents the successful evolution in early stages of the evolutionary process. External information is used to decide which modules have to be evolved in each particular stage, as well as under which evaluation function, that is, there is selective knowledge introduction. Due to the architecture modularity degree, groups of modules can be evolved in the same controller but under different evolutionary processes. Additional evolutionary stages evolve connections between the modules and obtain the final complete controller. This method is successfully applied in chapter 6 for the evolution of a walking behaviour in an Aibo robot coordinating 16 DOF with 28 devices.

Following the description of the progressive design method, a way of externally influencing the architecture is also shown. At times it may be required to slightly modify the evolved behaviour, based on the situation outside of the robot. Chapter 7 provides a method to slightly modify the behaviour of one controller based on the value of an external tonic signal. The external signal is received by the modules coming from a completely different controller (any external control process), and can be used to *tune* the architecture behaviour to the different situations that the robot is experiencing. One simple example will be to modify the speed of the Aibo walking pattern, as will be demonstrated in chapter 7.

In chapter 8, the way how the architecture functions on the inside will be analysed. In this chapter, it is observed that the modules that constitute the controller establish an inner communication system between them, which taken as a complete vector describes the current state of the robot on a higher level of description than the raw sensory information. The architecture facilitates the analysis of the robot's neural mechanisms, and in particular the identification of the neural substrates of the internal representation. This means that this complete internal vector describes similar situations that the robot is experiencing with similar values, even if the actual sensory values are different. They are performing a kind of semantic assignation of meaning to their experiences. Additionally, the vector state is easily accessible from the outside simply by looking at the modules output. This fact allows the direct observation of the

current ensemble state.

Based on these results, chapter 9 concludes the thesis by proposing a method to integrate deliberation within the architecture. By incorporating this method, a higher deliberative process would make use of the vector state to deliberate its current situation, decide what to do next, and communicate appropriate action's to the modules using the external control signal. It also includes a discussion about how the model could be useful for other purposes in AI robotics, as well as how the architecture can be used to shed light on the mind-body problem.

Additional appendices include additional information on how several parts of the thesis experiments were performed, from a practical point of view. Appendix A, for example, will detail the process of building the simulator and the transfer of information to the real robot system used in this thesis; the Aibo robot. Appendix B will explain how to interface the evolutionary algorithm with the simulator for offline evolution, and finally, appendices C through to F will include a detailed list of the results of all the experiments performed in chapters 4 to 8, respectively.

1.2.1 Original contributions

We have identified the following points as the main contributions to the thesis:

- Two different types of modularity in neural controllers have been identified and defined, though to date only one type of modularity has been used in existing literature related to modular neural networks for robot control (namely, the decomposition of a global task in terms of required sub-tasks). This thesis proposes a new type of modularity which is generated at the level of the robot's devices. It has been identified at which modular level each type of modularity works, and how they can be combined to produce extremely complex controllers in evolutionary robotics.

- Based on this analysis, a distributed neural architecture for the control of complex robots is defined. The architecture is completely distributed with no central controller, and provides a method for the control of complex robots with neural networks. The method provided performs an internal representation of perception, with this representation being easily accessible from the outside. Internal representation may be used by other processes for more deliberative purposes, though this point is only lightly discussed in this thesis.

- A staged method for the generation of behaviours in evolutionary robotics is introduced, allowing for a modular introduction of the required knowledge. Modular organization is important, as it means that the introduction of selective knowledge is possible, that is, the introduction of knowledge that may affect only a small part of the controller (as small as a single module, in fact). We call this process *progressive design*.

These contributions make our approach interesting from at least three points of view:

1. Evolutionary robotics: a novel modular conception based on hardware elements for the generation of behaviour in autonomous agents is defined.

2. Control engineering: a fully collaborative neural control architecture based on small modular units with internal representation of perception is developed.

3. Cognitive systems: a possible answer to the mind-body separation dilemma by using an internal robotics framework is postulated.

1.2.2 Summary of results

1.2.2.1 Related papers, in journals and books

- C. Angulo, R. Téllez, and D. Pardo, Internal Representation of the Environment in Cognitive Robotics, International Journal of Robotics and Automation, vol. 24, issue 3, ACTA Press, 2009

- R. Téllez and C. Angulo, Progressive design through staged evolution, Frontiers in Evolutionary Robotics, I-Tech, 2008

- R. Téllez and C. Angulo, Modularity in artificial neural networks, Encyclopedia of Artificial Intelligence, Information Science Reference, 2008

- R. Téllez and C. Angulo, Webots software review, Artificial Life, Volume 13, Issue 3, 2007

- O. Vilarroya and R. Téllez, La madurez de los Aibo, Palabra de robot, Publicacions de la Universitat de València, 2006

- L. Holh, R. Téllez, O. Michel and A. Ijspeert, Aibo and Webots: simulation, wireless remote control and controller transfer, Robotics and autonomous systems, Volume 54, Issue 6, pp 472-485, 2006

- C. Angulo and R. Téllez, Distributed Intelligence for Smart Home Appliances, Tendencias de la Minería de Datos en España. Red Española de Minería de Datos,Vol: p1-12, 2004

1.2.2.2 Related papers, in conferences

- R. Téllez, and C. Angulo, Embodying cognitive abilities: categorization, in the 9th International Work-Conference on Artificial Neural Networks (IWANN'2007). Published in Lecture Notes in Computer Science, Volume 4507, p 781-788, 2007.

- R. Téllez and C. Angulo, Acquisition of meaning through distributed robot control, in the Proceedings of the ICRA workshop Semantic Information in Robotics, Rome, 2007

- R. Téllez and C. Angulo, Tactical modularity for evolutionary animats, in the Proceedings of the 9th International Conference of the Catalan Association for Artificial Intelligence, Perpignan, 2006
- R. Téllez and O. Vilarroya, Towards an experiences based artificial intelligence (abstract), 50th Anniversary Summit of Artificial Intelligence, Monte Verita, Switzerland, 2006.
- R. Téllez, D. Pardo and C. Angulo, Comparison of synchronous and asynchronous control modes on dynamic control (abstract), First URBI Workshop 2006, Paris, France, 2006.
- R. Téllez, C. Angulo and D. Pardo, Evolving the walking behaviour of a 12 DOF quadruped using a distributed neural architecture, 2nd International Workshop on Biologically Inspired Approaches to Advanced Information Technology (Bio-ADIT'2006), Osaka, Japan, 2006. Published in Lecture Notes in Computer Science, Volume 3853, p. 5 - 19, 2006
- R. Téllez, C. Angulo and D. Pardo, Completely neural architecture for the general control of autonomous robots, European Symposium on Nature-inspired Smart Information Systems, Albufeira, Portugal, 2005.
- R. Téllez and C. Angulo, A distributed architecture for sensory-motor coordination (abstract), in the 3rd International Symposium on Adaptive Motion in Animals and Machines (AMAM'2005), Ilmenau, Germany, 2005.
- R. Téllez, C. Angulo and D. Pardo, Highly modular architecture for the general control of autonomous robots, in the 8th International Work-Conference on Artificial Neural Networks (IWANN'2005), Vilanova i la Geltrú, Spain, 2005. Published in Lecture Notes in Computer Science, Volume 3512, p 709, 2005
- R. Téllez and C. Angulo, Evolving cooperation of simple agents for the control of an autonomous robot, in the Proceedings of the 5th IFAC Symposium on Intelligent Autonomous Vehicles (IAV04), Lisbon, Portugal, 2004

1.2.2.3 Invited talks

- Distributed neural control for the Aibo robot, University of Yamaguchi, Japan, 2006
- Tactically distributed sensory-motor coordination, IDSIA, Switzerland, 2006
- Towards a society of mind for autonomous robots, Sony Computer Laboratory, Paris, France, 2005

1.2.2.4 Media coverage

- Radio interview, Radio Barcelona, 15-IV-2006
- Press review of the research, El Pais, 29-IX-2005

1.2.2.5 Awards

- Robots i evolució. Winner of the honourable mention award at the 1st ACCC competition for the spreading of scientific research, 2003.

1.2.2.6 Teached summer schools

- Aibo programming summer school, University of Vilanova, 2005
- Aibo programming summer school, University of Vilanova, 2004

1.2.2.7 Co-directed master thesis

- Distributed control of wheeled robots, by Javier Collado, Technical University of Catalonia, 2006

1.2.2.8 Stages

- Research stage at EPFL, Switzerland, 3 months in 2005 for the development of a tonic control of the Aibo walking

1.2.2.9 Co-organized workshops

- Emergent Behaviors and Cognition in Embodied Systems, IWANN Special Session, 2007

It is absolutely safe to say that if you meet somebody who claims not to believe in evolution, that person is ignorant, stupid or insane

Richard Dawkins

2 Evolutionary robotics

Chapter 1 stated our interest in using evolutionary robotics methods for the training of artificial neural networks. This second chapter is an analysis of the selected framework, and is organized as follows: firstly, techniques from the ER approach to be used for ANN training will be described in sections 2.1 to 2.4. In section 2.5 the limitations of the current ER framework for our particular application will be described, and finally, in section 2.6, a solution to the limitations described is proposed.

2.1 Introduction

Evolutionary robotics is a general framework for the automated design of controllers for autonomous robots. It reproduces in robot controllers the Darwinian principle of selective reproduction or natural evolution; by accumulating small changes [Dawkins, 1987] in a progressive development. ER uses genetic and evolutionary algorithms to develop control programs for autonomous robots. A robot control system is represented as an artificial chromosome that progressively updates throughout its evolutionary process until it successfully accomplishes the desired robot behaviour, in this way simulating genetics and natural selection laws. Since a robot controller is based on ANNs as processing elements, in this thesis artificial chromosomes will encode ANN weights, so that *network connections of ANNs are evolved*.

Evolutionary robotics tends to focus on processes for building task-oriented controllers rather than model-based controllers, and the designer is therefore not required to specify precisely how the controller should work in each and every potential situation [Nolfi and Floreano, 2000]. Furthermore, ER stresses the importance of an agent possessing a body, and that this body be situated in a physical environment. During the evolutionary process the artificial system autonomously develops its own skills through close interaction with its environment. Interaction with the environment makes it possible to evaluate how well or how poorly a robot controller, expressed in the form of a *genetic code*, is adapted for the task at hand.

Evolutionary robotics shares several similarities with the well known behaviour-

based robotics framework [Arkin, 1998] since the role played by the environment is extremely important in both approaches. However, while in behaviour-based robotics the engineer carefully decides beforehand on the generation of a set of basic behaviours associated with the robot task, in evolutionary robotic behaviours emerge as a result of the self-organizing process which is driven by a fitness function.

Generally speaking, the term *emergence* refers to behaviour that has not been explicitly programmed into a system or agent [Pfeifer and Bongard, 2007]. The evolutionary process evolves the required knowledge for solving the task through experience, and self-adjusts its control system to meet the specific demands of the domain. This means that if avoiding obstacles is a necessary component for good performance, the genetic algorithm will select those networks which are best suited to avoid obstacles [Gomez and Miikkulainen, 1996, Moriarty and Miikkulainen, 1996b].

2.2 The evolutionary process

Given a robot, a working environment and a task to be solved, the evolutionary process will attempt to find the most suitable robot controller with which to perform the required task in the specified environment. The evolutionary process achieves this goal by acting in the following way (see figure 2.1):

1. To begin with, the designer creates a starting population of chromosomes with initial random values. Each chromosome encodes a *possible* control system for the robot *in some manner*. These chromosome-encoding controllers are referred to as *genotypes*, and the specific manner in which a genotype encodes the robot control system is known as the *encoding scheme*. Encoding scheme selection is currently an intensely researched subject due to its high impact on the success or failure of the evolutionary process. From this initial step a pool of random potential control systems is therefore obtained.

2. Next, one control system is randomly selected from the pool of genotypes, and transformed into the control system of the robot. The resulting transformation of the genotype into a controller is referred to as the *phenotype*, and this transformation is called *genotype-to-phenotype mapping*. The phenotype is then installed in the given robot for its control.

3. The robot is then *evaluated* within the given environment. It will move around freely and manipulate its environment as specified by its controller. It will exhibit a behaviour according to what is specified by the controller, the robot body, and what it encounters in its environment. After a predetermined amount of time, referred to as *evaluation time*, the robot will be stopped. This is the most time-consuming stage of the evolutionary process, and the potential variation in the time required for this stage can be several orders of magnitude, making the design of this phase very

important. A good design can reduce evolution time from days to minutes, since thousands of tests could potentially be required for the evolution of just one single controller.

4. Task performance is *evaluated* by a cost function as defined by the designer, and is referred to as *fitness function*. Fitness function automatically calculates how well or how poorly the robot performs the required task. When the process is started the robot's behaviour can be very chaotic due to the genotypes having been randomly created, and the resulting fitness value may be very low. It is expected that the fitness value will increase as the evolutionary process progresses and new generations are tested.

5. Steps 2 to 4 are repeated for all the genotypes in the pool, and an ordered list with the score obtained by each genotype is generated. All the genotypes are usually tested more than once due to the random nature of the test, and the genotypes final evaluation score is then calculated as the mean average of its testing times.

6. Based on the ordered score list the best genotypes (*parents*) are allowed to be combined with each other by using genetic operators (duplication, crossover and mutation, amongst others), giving birth to a new population (*generation*) of genotypes (*offspring*). Offspring are added to the genotypes pool and the genotypes with the lowest fitness values are automatically deleted from the pool, ensuring a pool with the best possible genotypes.

7. Once a new generation of genotypes has been created the evolutionary process is repeated from step 2 for the number of generations defined by the designer. The evolutionary approach assumes that the mean fitness value obtained by one generation will improve as the number of generations increase, and a good controller for the task will eventually be obtained.

In a general evolutionary process, evolution begins with an initial random population of different solutions with a maximum diversity. As the process evolves, the random population will converge upon a solution, and the subsequent genetic diversity of the pool of solutions will decrease. At some point an equilibrium is attained, and no further increases in the fitness value are observed. This situation is known as *stagnation*, and marks the point where the diversity of the population is at its lowest value. At this point either the evolutionary process should be stopped, or the probability of the mutation operator should be increased.

Precisely, how to initialize the evolutionary method is yet another fundamentally important issue. Randomness when generating the first generation of genotypes can potentially imply that a large amount of time could be required before any useful behaviours for the task emerge. Furthermore, in the case of a very complex robot it may not be possible for the evolutionary process to generate a growing fitness process, due to the fact that all the initial genotypes obtain zero fitness values in every evaluation. A classic example of this is when

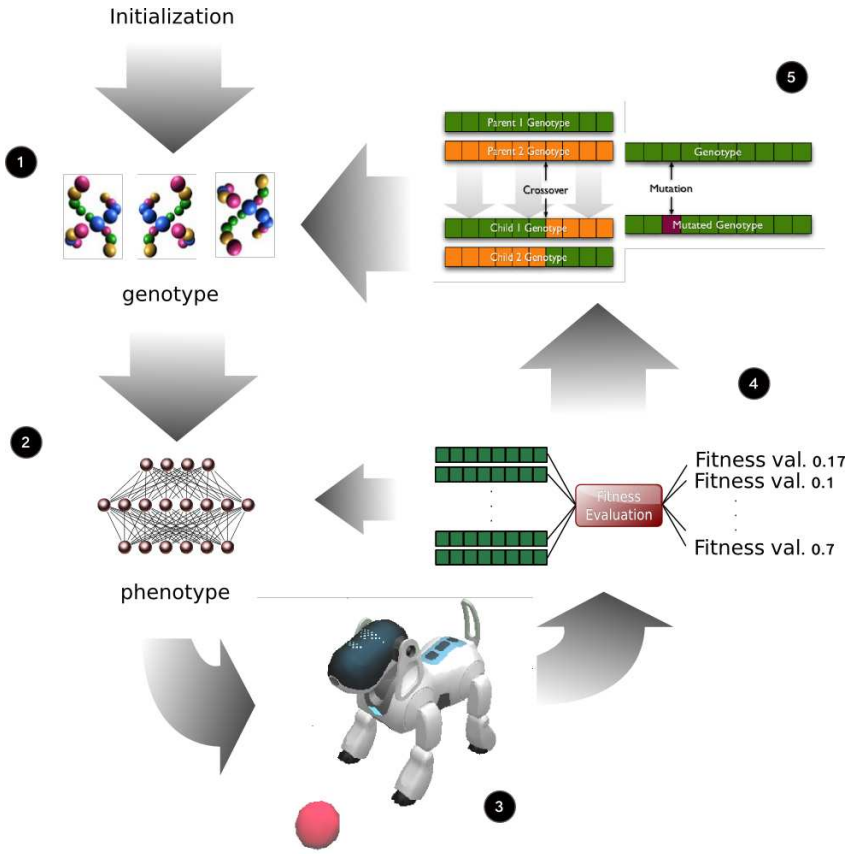


Figure 2.1: Schematics of the evolutionary robotics process

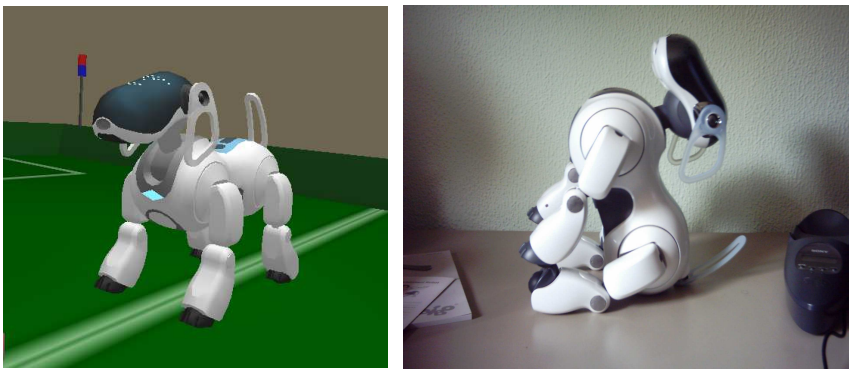


Figure 2.2: Simulated and real Aibo robot.

attempting to evolve the walking behaviour of a humanoid, and all the initial solutions result in the robot throwing itself to the ground, implying that their fitness is zero[1]. Hence, no improvement in terms of fitness value will be observed through generations, and the walking behaviour will not evolve. This problem is known as the *bootstrap problem* [Nolfi and Floreano, 1998], and though several solutions have been proposed the problem has yet to be entirely resolved (see section 2.5).

A third comment regarding the evolutionary process is that algorithms develop required behaviours for the task in a self-organizing process, that is, entirely free of human intervention. Acting in this way, fresh and non-intuitive (*unbiased*) good solutions can be found. Bias is understood as any type of knowledge that the designer includes in the fitness function that results in the evolutionary process driving its search towards a specific solution. As will be seen later, we firmly defend the fact that obtaining an unbiased controller is only possible under certain simple circumstances.

2.2.1 Evolution in simulated and real robots

Every time that a generation of controllers is tested in the evolutionary loop, the controller is loaded in the robot and the robot is left to interact with the environment designed for the task while running its controller. This interactive test step can be performed in both simulated and real robots, and each approach carries its own advantages and drawbacks. When done in simulation, the process is referred to as *off-line evolution*. Off-line evolution is used mainly because it is inexpensive, avoids the need to concentrate on robotic engineering issues, and above all it allows for a much faster evolution; what would take months in a real robot could take mere minutes in a simulator, since the entire evolutionary process can be automated and executed at high speed. The draw-

[1]Moreover, initial random behaviours can make the robot demonstrate behaviours that may threaten its integrity when the evolution is performed in a real robot.

back with simulations is that they cannot completely model the robot and its environment, - because no such thing as a perfect simulator exists. Hence, in some cases, results obtained in the simulator are not valid for the real robot, - particularly in cases where the genetic algorithm takes advantage of unmodeled aspects of the simulator. This type of situation could lead to problems when transferring solutions from the simulator to the real robot. Differences between the simulator world and real world are known as the *reality gap*, and this problem intensifies as the complexity of the robot, its task and its environment increases. In an attempt to resolve the reality gap problem, Jakobi defined the concept of *minimal simulation* [Jakobi, 1998]. This concept advocates that the simulation be strictly limited to those aspects of the robot and its environment that are absolutely necessary for the task at hand. The rest of aspects should be modeled as noise, resulting in the generation of a controller that does not depend on those features. The result of simulations that implement the concept of minimal simulation is a far faster simulation, (as they do not have to simulate everything), as well as a robust simulation when crossing the reality gap and transferring the controller to the real robot. It must be noted however that to model a simulation in such a manner is not at all a straightforward task.

Physical robots are used during the evolutionary process when the situation is so complex that it cannot be modeled in simulation. This process is referred to as *on-line evolution*. Its main drawback however, is that on-line evolution takes longer than off-line evolution by several orders of magnitude. Furthermore, additional effort is required to recharge batteries and fix robots. Due to these drawbacks, and in order to obtain the best of both worlds, a combination of the two methods can be used. In such a case a limited simulation of the robot and the environment is used in the first stage (usually using minimal simulation criteria), and the robot is evolved off-line, thereby generating a controller for the simulated robot. Following this, in the second stage, this controller is transferred to a real robot and, if necessary, it (the controller) is on-line evolved for several generations. The second evolutionary stage allows controller modifications to better prepare it for the real world. Present day simulators are suitable for this later approach with off-the-shelf solutions. One example is the commercial Webots simulator [2] developed by Cyberbotics. Webots provides both an already simulated environment as well as a solution for transfer to real robots for a long list of commercial robots [Michel, 2004, Téllez and Angulo, 2007]. This thesis uses both on-line and off-line evolutionary processes with Webots software. Additional information comparing simulation approaches and real-robot approaches can be found in [Walker et al., 2003].

[2] We have contributed to this program through the development of the Aibo robot simulation [Holh et al., 2006]. This software contribution was used in further experiments, described in chapters 4, 5, 6, 7 and 8. A complete description of our development can be found in appendix A.

2.2.2 Control architectures

A control architecture specifies how to organize the robot control programme. Within the evolutionary robotics paradigm, architectures can be found based on genetic programs [Koza, 1991], classifier systems [Holland, 1987], and neural networks [Nolfi and Floreano, 2000]. In principle, evolutionary robotics is capable of evolving any control architecture that can be replicated and mutated. The question is then; what would be the most suitable type of architecture to be used within evolutionary robotics?.

In the case of genetic programming, Lisp-like programmes were a very attractive option in the past due to the treelike structure, which ensures an especially robust application of genetic operators [Steels, 1994, Olmer et al., 1996, Nordin et al., 1998]. Classifier systems are based on a discontinuous set of discrete rules, and have been successfully applied in [Colombetti and Dorigo, 1992, Dorigo, 1995] to several examples. It is difficult however to demonstrate that this rigid system is the most appropriate for a complex robot.

Artificial neural networks on the other hand present some very interesting features for evolutionary robotics, and are therefore the preferred option of designers.

- ANNs provide a smooth search space in which to search for a suitable controller, and gradual changes in the parameters of the net, having been evolved, correspond to gradual changes in the controller behaviour.
- The same network structure can be used for controlling different robots performing different behaviours by simply altering the value of the network's weights accordingly. A good example of this are the experiments shown in [Nolfi and Floreano, 2000], which in most cases use a simple feedforward neural network, - even if the behaviours required from the robots are very different. Furthermore, this implies that the same encoding scheme can be used independent of the functionality of the control system.
- Neural networks allow for the smooth integration of other adaptive processes with artificial evolution, such as supervised or unsupervised learning [Floreano and Mondada, 1996, Nolfi and Floreano, 2000].
- Evolved behaviours are mainly low-level behaviours which require a close coupling between the sensors and the actuators of the robot. Artificial neural networks provide a straighforward mapping between sensors and motors [Husbands et al., 1995, Nolfi and Floreano, 2000]. Furthermore, they can easily accept continuous input signals, and generate either continuous or discrete output answers, depending on the transfer function chosen.
- Artificial neural networks are robust to noise, and oscillations in individual input values do not highly affect output generated by the net since output is a combination of several signals. This feature makes ANNs especially appropriate for use in real robot environments.

- Due to the methaphore they represent, ANNs allow the implementation of ideas derived from biology, making bio-inspiration a good source of inspiration for the generation of controllers [Billard and Ijspeert, 2000, Watson, 2002b, Nakada et al., 2004]. Moreover, the other way round is also valid, and controllers based on neural networks may well help to generate new hypotheses on the workings of biological systems [Ijspeert, 2001, Hallam and Ijspeert, 2003].

When the evolutionary process evolves ANN-based controllers it is referred to as *neuro-evolution*. Neuro-evolution is then the use of evolutionary algorithms to evolve neural networks that will later be used as controllers for robots. For robot control, evolutionary robotics is especially advantageous in that it does not require training examples for the evolution of the nets. In addition, evolutionary robotics has no restrictions whatsoever regarding the type of neural network to be used. Examples include: perceptrons [Floreano and Mondada, 1994, Lund and Hallam, 1996, Nolfi, 1997, Dorigo et al., 2004], continuous-time neurons [Hopfield, 1984, Yamauchi and Beer, 1994, Beer, 1995, Gallagher et al., 1996, Hallam and Ijspeert, 2003, Seys and Beer, 2004, Téllez et al., 2006], and recurrent neural networks [Floreano and Mondada, 1996, Nolfi S., 2000]. As a matter of fact, ANN structure can also be included in the evolutionary process [Gruau, 1994, Gruau, 1995, Whitley et al., 1995, Gruau et al., 1996, Gruau, 1997].

2.3 Encoding schemes

The encoding scheme is responsable for specifying the way the phenotype is encoded in the genotype. Encoding scheme selection is a vitally important issue since it can be used to drive the evolutionary search towards areas that are more likely to contain a solution [Stanley and Miikkulainen, 2003]. In most encoding schemes genotypes are made up of strings of values which represent the parameters of the phenotype to be evolved. These values can be expressed as either binary values or real numbers. There are a few exceptions however, and they encode other information such as primitives [Hornby and Pollack, 2002, Hornby, 2003, Hornby et al., 2003]. Genotypes can encode many of the phenotype's characteristics, including control architecture, robot morphology and even rules on how the genotype-to-phenotype mapping should be performed.

When applied to the evolution of neural networks, two classes of encoding methods exist: *direct encoding*; where the genotype encodes all the values of the connections of the neural net, those are, the weights, in a straight way, and *indirect encoding*; where the genotype does not encode the net weights, but a list of developmental rules, each rule describing how the network must be constructed. Additionally, a second type of classification can be made, based on size: encoding schemes for *fixed size* networks, and encoding schemes for networks that *vary in size*. Throughout this thesis we will use the simplest case of encoding, that is, direct encoding with fixed size, and encoded values will be strings of real numbers encoding the weights of the nets.

2.3.1 Genetic algorithms and genetic operators

In order to evolve a robot controller, a genetic algorithm and genetic operators must be selected. The genetic algorithm (GA) defines the steps that determine how the different processes interact, as well as how and when the genetic operators will be applied. Genetic operators define the rules on how to obtain the offspring genotypes from the parents ones.

The GA specifies how the process by which genotypes are selected from the pool of available genotypes to generate offspring works, as well as the process by which the new offspring are introduced back into the pool to replace the old ones. A good introduction to genetic algorithms can be found in [Mitchell, 1996]. GAs have been used for solving various different types of problems, with ER being only one of its applications. Several issues need to be taken into account when using genetic algorithms for evolutionary robotics :

- Performance of different genetic algorithms has been compared in their application to standard tasks like the pole-balancing problem described in [Gómez and Miikkulainen, 1999, Stanley and Miikkulainen, 2002] or the predator-prey games [Yong and Miikkulainen, 2001]. Results show however that there is no GA that is the best for all types of problems, and although certain GA perform better than others in the resolution of certain tasks, this does not imply that GA is the best solution for evolutionary robotics in a general sense.

- One difference between robotics and any other field employing GA is that robotics has to handle the noisy values that are provided by sensors or sent to actuators. This implies that one controller can behave differently on two different occasions, and subsequently obtain contrasting fitness values. A GA therefore needs to take this fact into account when selecting the optimal controllers for reproduction. This obstacle is generally overcome by testing each and every controller several times in different initial situations, and then calculating its fitness as a mean average.

Some interesting GA's for neuro-evolution include the pioneer SANE algorithm [Moriarty and Miikkulainen, 1996a, Moriarty, 1997]; Enforced SubPopulations (ESP) [Gomez and Miikkulainen, 1996, Gómez and Miikkulainen, 1999], and derivatives: multiagent-ESP [Yong and Miikkulainen, 2001], Hierarchical-ESP [Gomez and Schmidhuber, 2005]; NEAT [Stanley and Miikkulainen, 2002]; GENITOR [Whitley et al., 1993]; or EVOLINO [Wierstra et al., 2005], to name but a few. The ESP algorithm will be used in the experiments described in this thesis. Additional genetic algorithms can be found in [Barto et al., 1983, Anderson, 1989, Watkins and Dayan, 1992, Pendrith, 1994, Whitley et al., 1995, Gruau et al., 1996, Sutton and Barto, 1998, Meuleau et al., 1999].

Genetic operators describe the way in which new offspring genotypes are obtained from parents. Genetic operators are responsable for the genetic diversity of the population as well as the convergence rate. One of the more commonly used operators is the *crossover* operator, which combines two parent genotypes

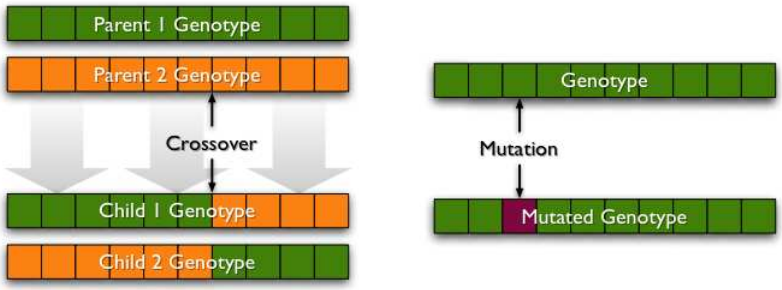


Figure 2.3: Crossover and mutation operators

with two offspring by selecting a random point along the genotype around which genetic material is swapped between the two parents. Another typical operator is *mutation*, which introduces random changes into the genotype. This operator is usually used with a low probability value that decides whether any element of the genotype needs to be changed after crossover. Changes introduced by the mutation include the switching of its value for binary encodings, or the addition of a small amount randomly distributed around zero for encoding with real numbers.

2.3.2 Fitness functions

The *fitness function* rewards a controller based on the performance of the robot running such a controller while attempting to solve the problem at hand. It is an evaluation score of the controller's performance. Evaluation is condensed in the form of a mathematical function that calculates how well or how poorly the robot has performed the task (using the present controller), using available data such as the status of both the robot and the environment.

The way a fitness function is designed depends on the robotic setup for the task. If the evolutionary process is performed in simulation then all the parameters that may be required for measuring the correctness of the controller are available, - since everything can be measured. For real robot evolution fitness evaluation can be more complex, as hardly any parameters are directly available to automate the process. In general, there is no principled way of designing a fitness function, and its generation therefore entails a *trial-and-error* process.

Evolutionary robotics aims for fitness functions specifying a high level description of the required robot task. Fitness should include as few clues as possible as to how to achieve successful behaviours for this task. Fitness function defines the task that one wants the robot to complete, but not how it is to be performed. This freedom has in many cases lead to remarkably interesting and surprising solutions; for example in the evolution of walking quadrupeds, a technique was evolved that allowed the robot to throw itself through the air in order to advance [Reeve, 1999].

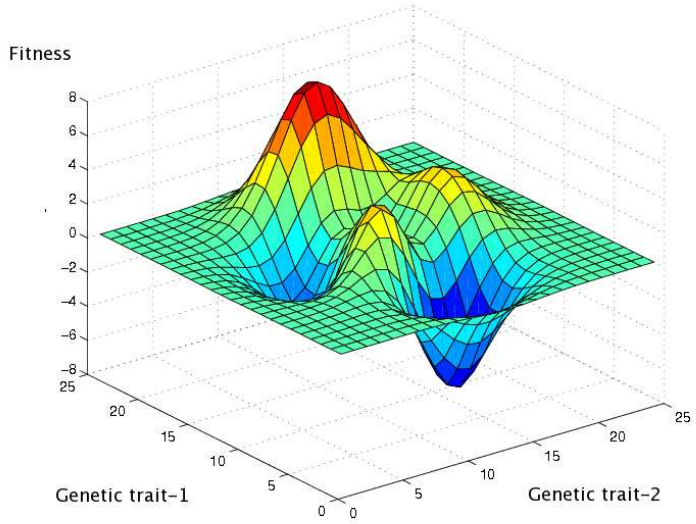


Figure 2.4: Three dimensional representation of a fitness landscape.

The *fitness landscape* is a representation of the search space of an evolutionary algorithm, given a fitness function. It associates fitness values with corresponding genetic traits (see figure 2.4). The dimensionality of this landscape corresponds to the number of parameters to be evolved. It can also be called the *search space*, referring to the fact that the evolutionary algorithm is moving through that space whilst searching for the most suitable combination of genetic traits to obtain the optimal fitness value.

When the robot is simple, the algorithm can progressively find solutions which are a little closer to the desired solution in each consecutive evolutionary step . Each evolutionary step builds upon the previous one, thus continually obtaining better solutions. However, in cases where the fitness landscape is massive due to the complexity of the robot or the task, an algorithm cannot find partial solutions to solve the task, and can therefore not improve through succesive generations (better known as the bootstrap problem, as defined in section 2.2). In these cases, a possible solution is to add terms to the fitness function; based on the additional knowledge provided by the designer. These added terms act as the drivers of the evolutionary process, indicating where a partial solution may be located. They can also smooth out the fitness landscape. The addition of fitness terms provides clues for the algorithm to find the solution and restricts the search space to regions that the designer knows contain a solution. When this sort of situation occurs, the fitness function is said to contain bias.

While evolutionary robotics looks promising, it is not the golden solution to

robot control generation. In chapter 6 it is postulated that the introduction of bias into the fitness function is unavoidable if so many constraints are imposed on the evolutionary process. These constraints include: the morphology of the robot, the task to be solved and the environment.

2.4 Improving the initial evolutionary methodology

Several modifications to basic evolutionary processes have been proposed by different researchers to improve basic evolutionary methods and also to allow the evolution of more complex behaviours. We have listed a few of them here.

2.4.1 Incremental evolution

Incremental evolution (or *incremental learning*) is a technique that proposes to evolve neural network training by performing successive teaching steps with an increase in the complexity of the task being taught on each successive step. The use of incremental evolution has been proposed as a solution to reduce large search spaces [Elman, 1991, Gomez and Miikkulainen, 1996]. The network controller is first trained to perform a simple task; t_1 related to the final global task t_{goal} but in a simpler form. When the network has learnt the initial task, a second task t_2, more complex than t_1, though still simpler than the goal task t_{goal} is then taught to the network.

These steps are performed until the final global task t_{goal} is successfully learnt. It is worthwhile to note that in incremental evolution there are two different kinds of tasks: *evaluation tasks*, which are used to evaluate the network's reproductive fitness ; and the *goal task*, which the network is evolved to perform. The goal task is the culmination of a series of less demanding evaluation tasks $\{t_1, t_2, ..., t_{goal}\}$, where the set of tasks is ordered from simpler to more complex. This form of learning can be considered as an appropriate model of *continual learning*.

Each evaluation task to be learnt by the neuro-controller is expressed in terms of a different fitness function. Hence, after an evaluation task has been mastered by the controller the fitness function is changed to that of the next evaluation task. This approach has worked well with a simple robot performing both simple and complex tasks like in [Gomez and Miikkulainen, 1996, Gómez and Miikkulainen, 1999, Yong and Miikkulainen, 2001, Islam et al., 2001], though successful application of this approach for use in a complex robot has not yet been reported.

In [Doncieux and Meyer, 2004] an alternative to incremental evolution has been proposed. Rather than using the existing knowledge on the task to evolve the generation of a set of tasks with increasing complexity, the authors proposed the use of knowledge to guide the evolution in two directions: firstly, the evolutionary process evolves modules instead of neurons and connections, and secondly, information is provided to suggest to the algorithm which connections

are likely to be useful in the final controller. This method has been successfully applied to the control of a simulated lenticular blimp.

More recently, new approaches based on the basic incremental evolution idea have been presented, like the use of exaptation [Graham and Oppacher, 2007, Mouret and Doncieux, 2009a], multi-subgoal evolution described in the work of [Mouret and Doncieux, 2008] or behavioral robotics [Mouret and Doncieux, 2009b].

A kind of incremental evolution, divided into stages, will be used in this thesis.

2.4.2 Co-evolution

In most evolutionary robotics works, the evolved controller is used to guide a single robot or agent. When several agents working together need to be controlled, it becomes necessary to use a co-evolutionary method, i.e. to evolve a controller for different roles in a common task, - either in a single population for all the agents, or in multiple isolated populations [Yong and Miikkulainen, 2001]. In *co-evolution*, controllers for two or more robots are evolved simultaneously in an open-ended manner whilst they interact with each other. Interaction will imply that changes in the behaviour of one robot drive further adaptation in the other.

Species interact with each other within a shared domain model in either a cooperative, or a competitive relationship. Co-evolution therefore evolves a group of agents in order to show them how to cooperate or compete to achieve a common goal while every agent has its own and different vision of the whole system. In *competitive* co-evolution, the role of each agent is against the role of the other agents (one agent's loss is another agent's gain), while in *cooperative* co-evolution agents share rewards and penalties for successes and failures. In competitive co-evolution, each agent receives its own personal score based on a fitness function which is particular to each agent. However, if a cooperative controller is required, a training algorithm will reward all the agents with the same score. A single fitness function will usually be used to reward all the agents.

Problems in which the solution can be divided into subparts are the best suited to make use of co-evolution. Subparts are evolved within their own population, each contributing as best they can to the final solution. Every subpart interacts and cooperates with the others in a combined effort to most effectively solve the problem.

The most widely studied case of co-evolution is the *predator and prey* game [Koza, 1991, Reynolds, 1994]. In this game, one robot, - acting as the predator, tries to capture the other robot - which acts as the prey. The prey uses evolutionary procedures to generate better solutions to avoid the predator, whilst the same evolutionary process simulaneously tries to generate better catching strategies for the predator. This co-evolutionary process will produce increasingly difficult challenges for each of the robots, while also progressively increasing the performance of each of the robots. This phenomenon is known as the *arms race*, [Dawkins and Krebs, 1979] and may lead to highly proficient robots

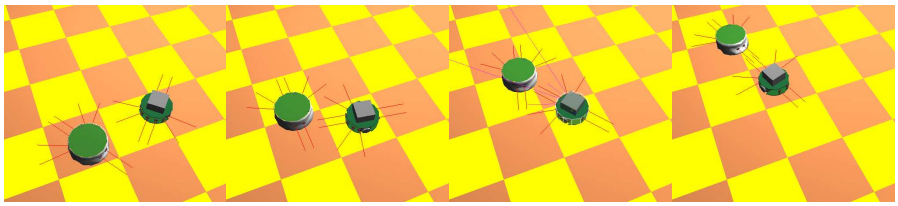


Figure 2.5: Sequence of the predator and prey game where two robots were co-evolved in a competition. The robot with the green top (the prey) tries to avoid the robot with gray box on top (the predator).

of both types [Nolfi and Floreano, 1998]. It cannot be guaranteed that this process will indefinitely increase the robots performance however, as cycling loops may appear (this refers to when the robots repeatedly cycle through the same solutions). In [Floreano et al., 2001] it is suggested that co-evolution may help to solve the *bootstrap problem*, that is, the inability to gradually increase the performance of the controller due to the complexity of the task.

Several examples of co-evolution for the evolution of a simulated predator and prey game can be found in [Yong and Miikkulainen, 2001]. Real Khepera robots were used in [Floreano et al., 2001] for the evolution of the same game. In [Ostergaard and Lund, 2003] robots were co-evolved to play robot soccer, -even competing with an opposing team. In [Potter and Jong, 2000] co-evolution was used to evolve co-adapted subcomponents. Their co-evolutionary architecture is called cooperative co-evolution, and consists of the evolution of different subcomponents, each one with its own population, which interact in a shared environment in order to solve a specific task. The main interest of this architecture is that the number of subcomponents required for solving the task is not determined beforehand, but that it emerges based on a stagnation criteria. Unfortunately the authors have only applied this architecture to the resolution of numerical problems. The same principle of subcomponents will be used in chapter 4 of this thesis for the evolution of the DAIR architecture.

2.5 Evolution of ANNs in complex robots

The focus in this thesis is on the use of evolutionary robotics for the generation of controllers in complex robots, which is to date an unresolved issue. We define the *complexity of a robot* by the number of sensors and actuators that have to be coordinated, and it follows then that the complexity of a robot increases correspondingly as its number of devices increase (sensors and actuators). This definition is based on the description of a *complex system* provided in [Simon, 1969](page 218, 2006 Spanish edition):

> [...], by complex system I understand the system composed of a big number of different parts which maintain a series of interactions between them.

This thesis focuses on neurocontrol in complex robots, that is, robots with dozens of devices. However, we observe that ER currently only achieves good results for simple (mainly wheeled) robots, that is, robots with less than ten devices (see all the examples shown in [Nolfi and Floreano, 2000]). The aim of this thesis is to provide a practical solution for complex robots controlled by neural networks; using evolutionary robotics as the training method. To date, only a small number of researches have applied ER to complex robots, and the solutions found are very limited and particular to the robot used. This thesis searches instead for the most general (which can be applied to any robot) and unbiased (which can find novel solutions) possible method. The reason for failing to apply ER to complex robots is primarily that the search space that the evolutionary algorithm has to face is huge (due to the high number of parameters to evolve; this dictated by the number of devices to control). Hence, firstly, it is difficult to find a solution through small changes, and secondly, the bootstrap problem prevents the generation of simple solutions with minimum fitness values at the beginning of the evolutionary process that could guide the evolutionary path towards the final solution.

Several solutions have been proposed to generate complex behaviours in complex robots, though none to date are entirely satisfactory. The use of incremental evolution to reduce the large search space has worked well for complex tasks in simple robots [Islam et al., 2001], but successful use in a complex robot has not yet been reported.

In [Nelson et al., 2002, Nelson et al., 2003b, Nelson et al., 2003a] the use of a fitness function with two modes was proposed: the first mode rewards the controller when it is not able to complete the task by using a subjective measure (completely determined by the designer) of the uncompleted task. The second mode only provides a reward when the task is achieved. A similar approach was also proposed in [Nolfi, 1997] for the generation of a garbage collector controller, or for the generation of a hand able to grasp things [Bianco and Nolfi, 2004]. The approach worked well in each case, given that all the robots were wheeled robots. As will be shown in chapter 6, this type of approach doesn't work when the first mode is not easily scorable (due to the complexity of the robot).

In [Doncieux and Meyer, 2004] an alternative to incremental evolution to guide the evolutionary process, - based on using the knowledge that the designer has about the task to evolve, has been proposed. The authors successfully applied their method to a blimp with 12 devices, even if the application of the same method to more complex robots seems difficult. Experiments performed in chapter 6 with an Aibo robot willing to walk have constated this point, showing that even allowing as much guidance as needed for this Aibo experiment, it was still impossible to evolve even simple initial solutions if only a straight monolithic approach was used.

According to the analyzed approaches, the bootstrap problem is the biggest challenge when dealing with complex robots, and only partial solutions based on special tricks for the particular controller to evolve exist. An example of this is a walking behaviour is evolved for a biped in [Mojon, 2004]; initializing the weights of the nets with small values, and allowing the robot to obtain a

few controllers that don't fall down at the first generations. In Reeves' work, [Reeve, 1999] walking quadrupeds have been evolved using symmetry to reduce the search space and avoid useless initial controllers.

Another solution that has been proposed to solve the bootstrap problem is co-evolution [Nolfi and Floreano, 1998], but no applications for complex robots currently exist, as was pointed out in [Doncieux and Meyer, 2004].

A possible solution for both problems is the simultaneous evolution of both the robot body and the neural controller [Sims, 1994, Funes and Pollack, 1999, Lipson and Pollack, 2000, Hornby and Pollack, 2002, Bongard and Pfeifer, 2003, Muthuraman et al., 2003, Muthuraman, 2005]. However, these approaches lack the ability to drive the evolutionary path of the body towards a predetermined body structure. Furthermore, these approaches usually make use of small errors introduced in the evolutionary system, generating physically useless robots.

By introducing restrictions into the fitness function that are based on previous knowledge of the situation, possible morphologies to be evolved can be constrained to the ones that are interesting for the given robot body. As was shown in [Muthuraman, 2005] this is a good approach for the complexification of controllers, though even if the author applied it to the evolution of other tasks it remains difficult to see how it could be applied to any robot for any task.

Ontogenic approaches to the evolution of morphology and control system also appear promising for the evolution of a complex behaviour in a complex robot [Bongard, 2003], even with its lack of directness of the evolutionary process towards a given and fixed body.

2.6 Proposed solution

From our point of view that there are several promising looking methods for the resolution of the control in a complex robot problem, namely: development of new encoding schemes, body evolution, use of learning, and design of new architectures.

New architectures adopt an engineering approach to the problem, and may be better adapted for robots once a suitable architecture has been engineered. After a few preliminary studies, and results as promising as those obtained in [Angulo and Téllez, 2004, Téllez and Angulo, 2004], it was decided to study the problem of architecture design. The solution has to be universal in terms of both robot structure and task to solve.

2.6.1 Use of selective bias

Whilst searching for a solution for the evolution of complex neural controllers, it was observed that to date no report of the evolution of a neural controller in a complex robot without using human knowledge in the process exists, that is, without a bias being introduced by the designer. Even though bias is not usually desired in ER, it does seem unavoidable. This research suggests that this

necessity for bias arises when the artificial evolutionary algorithm tries to reproduce the effects of natural evolution under unfair conditions. While natural evolution gradually evolved at the same time as the animals body plan, sensors and actuators, nervous system, and even environment and needs, artificial evolution tries to evolve the nervous system of a robot for a fixed given body, given group of sensors and actuators, and within a given complex environment and particular task required. A similar observation was stated in [Simon, 1969] (page 57, 2006 Spanish edition):

> Species within an ecosystem adapt to an environment where other species evolve at the same time. The evolution and future of those systems can only be understood from the knowledge of their histories.

It is observed that under those same conditions, and if the robot is complex, the evolutionary process is unable to find a good enough controller within the huge search space. Hence, we claim that bias is necessary under those tight evolutionary conditions. Bias can be introduced in the process to act on different parts, the most common being the introduction of information in the fitness function. The introduction of information in the fitness function drives the evolutionary search towards a subspace where the designer assumes there is a solution, acting as a reductor of the search space dimensionallity. This is the preferred place to introduce bias, since it is very simple to modify in case the evolutionary results are not as good hoped for.

The introduction of knowledge into the fitness function does not, however, guarantee the evolution of a good enough controller. Incorporating too much human design in an evolutionary process may end in the solution implemented not being that good from a generalist point of view. Partial solutions to this problem, described in the previous section, also attempted to introduce bias into the process (some of them in the fitness function). Unfortunately the solutions were not completely successful as the fitness information affected the entire controller. We propose instead the use of a mechanism where the controller is evolved using different fitness functions, each one affecting only selected parts of the controller. Each fitness function will contain the information bias that should affect only that part of the controller. The bias is therefore, selective in its affection. The controller has to be developed in several stages, and each stage will have separate parts involved in the process. We refer to this process as *staged evolution*.

2.6.2 Staged evolution

The proposed evolutionary process is defined as *staged evolution*. In staged evolution, incremental evolution is used at each stage to modify both the fitness function and the neural structure to be evolved at that stage. This implies that every stage will have a change in both fitness function definition and in controller's shape.

In [Hallam and Ijspeert, 2003], a form of staged evolution for the generation of a controller for both a lamprey and a salamander was implemented. They used blocks of neural nets to control the locomotion of a swimming lamprey, and their lamprey model was then later improved for the swimming and walking behaviour of a salamander [Ijspeert, 2001]. In a nutshell, they used the biological concept of central pattern generator (CPG) for the generation of oscillatory patterns in the robots. In both cases, a whole bunch of CPGs were created which they called segmental oscillators. Each segmental oscillator is an independent oscillator composed of several neural networks, and about 100 segmental oscillators are generated for the control of each of those animats. The output of some of these oscillators are connected to the muscles of the animat. Oscillators are also interconnected in order to synchronize their oscillations with a propagating wave which would allow the animat to swim and/or walk. Their method for the generation of the controller is divided into stages: in the first stage, a single segmental oscillator is generated using a genetic algorithm for the evolution of the weights (the number of neurons and their connections were already decided beforehand). Once the oscillator is created it is copied 100 times, and connections between neighbours established. These connections weights are then evolved for the generation of the final locomotory pattern. In the case of the walking behaviour of the salamander, an additional stage was added which allowed the evolution of two additional CPGs for the control of the legs.

In a similar manner, two separate and independent neural controllers were evolved for a Khepera robot in [Lara et al., 2001], - each one in charge of one different behaviour. Each module was also separately evolved using an evolutionary algorithm called ENS^3 [Pasemann, 1998] which allows for the automatic evolution of the structure and size of the network. One network was evolved to generate an obstacle-avoidance behaviour, and the second one was evolved for a light-following behaviour for a Khepera robot in a simulator. Once each behaviour was obtained, an additional step was used to *merge* them into one single controller. The same algorithm was used to evolve the connection between modules. In this case, both previously evolved modules were no longer evolved, and only new neurons which act as an interface between modules were allowed to be created . The final controller produced a robot that exhibited a joint behaviour of light seeking and obstacle avoidance. This architecture also presented a graceful degradation of the behaviours when the interface neurons were removed one at a time.

For the implementation of a staged process it is necessary to have some kind of structure that allows the division of the controller into different parts, and for each to be evolved in different stages. Previous approaches included this characteristic, but they were very tied to their own particular implementation. We propose the creation of a kind of universal structure (an architecture) suitable for any robot and any task. These requirements unavoidably lead to the use of modular neural networks, and due to the complexity level required, a powerful way to solve the problem is by using the *divide and conquer* principle. Hence, we advocate the use of a modular architecture that allows the progressive evolution of complex behaviours. To this end, a complete review of the modular

neural approaches to robot control will be introduced in the next chapter.

2.7 Conclusions

Evolutionary robotics is an excellent technique for the generation of controllers for autonomous robots. It stresses the importance of embodiment and environmental interaction for the generation of intelligent behaviours, and has been successfully applied to several robots performing different tasks, though the field still faces some tough challenges (for a list of them see [Mataric and Cliff, 1996]). Foremost amongst these, is the fact that it still requires improvement on multiple levels if complex behaviours in complex robots are to be evolved. We propose that in order to evolve behaviours in complex robots, two different approaches need to be used: introduction of selective bias in the evolutionary process, and evolution of the entire controller in stages.

In this thesis we will focus on the creation of a suitable architecture for use within this framework for any type of robot, wholly independent of the number of sensors and actuators it is composed of. If selective bias and staged evolution need to be implemented, then the best way to handle this problem is by using a modular neural architecture. In the next chapter we will review the available modular neural network-based architectures for intelligent robot control, in order to set out the current state of modular architectures and how evolutionary robotics can be improved by using them.

Modularity is a financial force that can change the structure of an industry

Baldwin & Clarck, 2004

3 Modularity in robot controllers

The previous chapter pointed out that a modular neural controller may be required in order to achieve the evolution of a neural controller in a complex robot solving a complex task. This chapter explores the coherence of such a suggestion, and subsequently determines that it is indeed a good way of structuring controllers. Current state of the art modular neural controllers are then reviewed in order to determine how the proposed DAIR architecture should ideally be structured.

In the 3.1 section an introduction to the concept of modularization as well as a justification of the benefits of using modularity in neural controllers is provided. Section 3.2 provides a taxonomy of neural modular architectures; outlining the applications of those architectures to robot control. Section 3.3 concludes that while some modularization applications serve as a point of departure, none of the architectures available nowadays will completely be general enough to be applied to any robot for any task. The central reason for this is that the core concept behind most of the existing modularization architectures is that of division by behaviour. This route will lead us to the next chapter where an entirely new level of modularization is proposed, giving birth to the DAIR architecture model.

3.1 Introduction

In this section we will introduce a few ideas concerning the concept of modularity and its application to the creation of intelligent autonomous robots. When creating controllers for the global behaviour of a robot, one can establish an initial classification in the following controller architecture terms: the *monolithic approach*, where a single module contains the complete required behaviour of the robot, and the *modular approach*, where the global behaviour associated to the task is decomposed into a set of simpler behaviours; each one implemented by one module.

Monolithic controllers implement all the required mappings between the sensors and actuators of the robot in a single module. The main advantage of this approach is that it is neither necessary to identify the sub-behaviours required

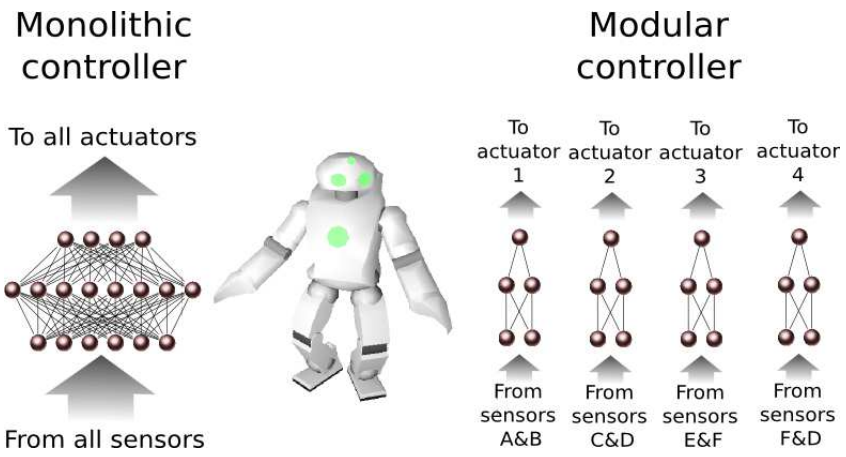


Figure 3.1: Differences between controlling a humanoid robot with monolithic (left) or with modular (right) controller: in the monolithic case, a single module takes care of all aspects of the robot, while with a modular approach a module can be designed for each activity, as well as a coordination mechanism between modules. When the robot to control is complex a modular controller is simpler than a monolothic one in terms of number of connections to train.

for the controller nor the relations between them. However, the fact remains that when the controller to implement is very complex in terms of the number of behaviours, sensors and actuators, it may in practice be impossible to create such a controller without experiencing great interferences between its different parts. In this case, the result is more often than not a controller that performs its assigned task very poorly.

In such cases a modular controller may do the job, dividing the monolithic complex controller into smaller, simpler parts. This is a direct application of the *divide and conquer* engineering principle. Furthermore, studies performed point out that complex behaviour cannot be achieved if modularity is not introduced at some level [Boers and Kuiper, 1992, Azam, 2000]. Either way, several experiments have already been conducted comparing the differences between the two approaches, in each case revealing that a modular controller outperforms a monolithic one [Nolfi, 1997, Auda and Kamel, 1997b, Ziemke and Bodén, 1999, Di Ferdinando and Parisi, 2000, Téllez and Angulo, 2004] in one way or another.

Before delving further into the list of modular architectures, some basic concepts such as *modularity* and *module* will be defined. The existing evidences of the necessity of modularity for complex control systems will then be pointed out, and a list of the advantages of such an approach provided.

3.1.1 Definitions

We will understand *modularity* as the property that has some complex computational tasks divided into simpler subtasks[1]. In such cases, the resolution of the complex task will be tackled by first solving the easier subtasks with the use of specialized computational systems, referred to as *modules*. The solution for the complex task is then obtained by *combining* the solutions provided by the modules [Azam, 2000]. In other words, modularity can be seen as the existence of a subset of variables in a system which can be optimized independently of the rest of variables in the system [De Jong et al., 2004]. With both definitions; when modularity is applicable to the resolution of a problem the problem is said to be *modular*, which implies the existence of a structure within the problem to be solved that the modules are able to capture.

In modular systems each of the modules primarily operates according to its own intrinsically determined principles. Modules within the system are closely integrated but remain relatively independent or *dissociable* from other modules. When the interactions between modules are weak and modules act independently from each other the modular system is known as a *nearly decomposable,* type, as defined in [Simon, 1969]. The same concept was later named the *separable problem* in [Watson et al., 1998]. This type of modularity is by far one of the most widely studied, and can be found everywhere, from business systems right through to biological systems [Simon, 1969]. In *nearly decomposable* modular systems the final optimal solution of a global task is obtained as a combination of the optimal solutions of the simpler ones.

However, the existence of decomposition in one problem doesn't imply that the sub-problems are completely independent from one another. In fact, a system may be modular and still have interdependencies between modules, as indicated in [Watson, 2002a]. Hence, a *decomposable problem* is defined as a problem that can be decomposed into further sub-problems, but where the optimal solution of one of those depends on the optimal solution of some of the others [Watson et al., 1998]. How this concept of a decomposable problem is applicable to the DAIR architecture will be clearly demonstrated, thereby helping to define it.

Some recent works [Husken et al., 2002, Callebaut, 2005] suggest that the strength or weakness of interactions between modules can be seen as a matter of degree. On the other hand, the resolution of decomposable modular systems is more difficult than a typical separable modular system, and is usually treated as a monolithic one in literature [Azam, 2000].

Most of the works that use modularity make use of the definition of a module given by J. Fodor [Fodor, 1983, Carruthers, 2005], which is very similar to the concept of *object* in Object Oriented Programming (OOP): a *module* is a domain specific processing element, which is autonomous and not capable of influencing the internal working of other the modules. The only way a module can influence another is by its output, this is, the result of its computation. Mod-

[1] Remember that in this thesis the word *behaviour* will be used in a general way along with the word *task*, indicating in both cases the same concept of robot purpose of creation.

ules are not aware of a global problem to solve or global tasks to accomplish, and are specifically stimulus driven. The modular system's final response to the resolution of a global task is provided by the integration of the responses of the different modules by a special unit. The specific control architecture that has been selected defines how this integration is performed. The integration unit must decide how to best combine the modules' outputs and produce the systems final answer, and is not permitted to feed information back into the modules. This definition of a module allocates space for both types of modularity (nearly decomposable and decomposable).

Weaker definitions of module exist, such as that proposed in [Carruthers, 2004].

3.1.2 Evidences for use of modularity

The interest in using modularity for robot control arises from the realization that modularity is an ubiquitous organizational principle found everywhere, and at all levels in natural and artificial complex systems [Simon, 1969, Callebaut, 2005]. From both biological and philosophical points of view it would appear that the use of modularity is a requisite for complex intelligent behaviour. Furthermore, from an engineering point of view, modularity seems the only way to deal with complex systems for complex tasks.

3.1.2.1 Biological evidences

There is physical evidence of the brain being a distributed, massively parallel and self-organizing modular system, which suggests that the brain is composed of very specialized modules which work more or less independently from one another, and are organized in a hierarchical structure with different levels and types of modularity; including parallel and serial modularity [Arbib, 1995]. Evidences include sparse connectivity between neurons found in the brain, where groups of networks form clusters; or functional modularity found in the vision system, with specialized modules for contrast, motion and colour detection. Further evidences can be found in [Caelli et al., 1999].

In [Ballard, 1986] it is suggested that due to the limited number of neurons that the brain contains, it was forced to evolve a modular architecture. Other authors suggest that competition in the brain between different networks results in certain networks becoming specialised for particular functions [Geshwind and Galaburda, 1987]. In this sense, a similar result has been reported in [R. Calabretta and Wagner, 1998], where a control architecture evolved specialized modules in competition with others for a robot controller.

3.1.2.2 Philosophical evidences

Philosophical arguments indicating that the mind has a modular composition have been brought to light by several authors. Some philosophers, such as Fodor, argue for a computational theory of the mind, where the mind is composed of computational modules but only at a *peripheral* level: a module for

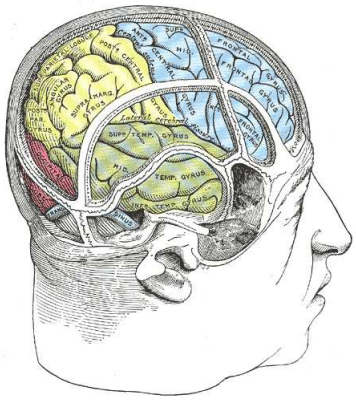


Figure 3.2: Brain modularity from a biological point of view: different zones of the brain are dedicated to different processes (free image obtained from Wikipedia).

vision, another for language, another for reasoning [Fodor, 1983, Fodor, 2000]. Other authors argue that mind modularity would reach deeper levels, created and shaped by natural selection [Pinker, 1997, Sperber, 2002, Weiskopf, 2002, Carruthers, 2004, Carruthers, 2005]. They are supporters of what has been called the *massive modularity hypothesis*, which claims that mind modularity is produced in all levels of mind. This means that the module for vision will itself be modular too, as well as all the other modules for the different activities. In this line of thought, artificial intelligence pioneer Marvin Minsky proposed in [Minsky, 1988] a kind of massive modularity theory in the form of a society of small mindless processing units, referred to as *agents*.

3.1.2.3 Engineering evidences

The principle of divide and conquer is a strategy commonly used in engineering for the resolution of complex problems. It consists of dividing the complex problem into a set of simpler sub-problems, and then combining the individual solutions of each sub-problem for the generation of the solution of the complex problem. This principle is used all the time in all types of engineering, and so its application to the design of complex robot controllers would appear to be sound idea too.

3.1.3 Advantages of modular approach

Aside from the evidences of the existence of modularity in natural and artificial systems, the use of modularity provides the following advantages for the resolution of complex problems:

1. A complexity reduction of the task to be solved [De Jong et al., 2004].

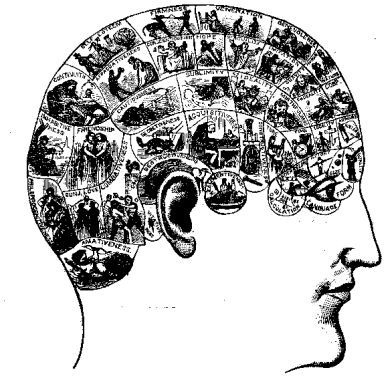


Figure 3.3: Massive modularity of mind in the form of a society of small mindless processing units (free image obtained from Wikipedia).

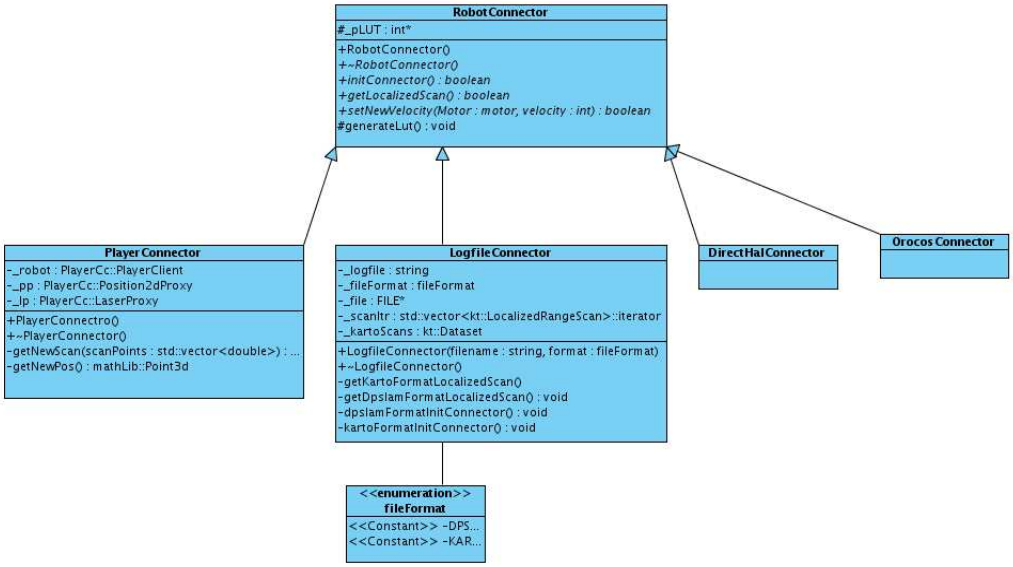


Figure 3.4: Engineering modularity. One typical example is the process of modularization in software development. C++ design based on object-oriented programming pursues this philosophy.

Whilst the optimization of all variables in a monolithic system is performed simultaneously and results in a large optimization space, in modular systems the optimization is performed independently for each module, resulting in a reduced searching space.

2. Scalability, in the sense that the use of modules should allow the resolution of more and more complex problems by using the modules already created for the creation of the new ones, or simply by adding new ones to those already existing [Urzelai et al., 1998].

3. Robustness. In modular systems, damage to one module affects that module alone, resulting in a loss of the capabilities of that module, but the whole system will still partially keep functioning [Simon, 1962].

4. Computational efficiency. If a processing task can be divided into separate and parallel subtasks, then the computational effort will be reduced in terms of both time and processing complexity [Azam, 2000].

5. Modularity may lead to meaningful representations of the system behaviour [Calabretta et al., 2000, R. Calabretta and Wagner, 1998].

6. Modularity may be a solution to the problem of *neural interference* especially for neural network based systems [Di Ferdinando and Parisi, 2000]. Monolithic networks suffer from the phenomenon of interference, also known as *crosstalk* (as described in [Jacobs and Jordan, 1993]). It is produced when either an already trained network loses a part of its knowledge when it is retrained for a different task (also known as *temporal crosstalk* [Jacobs et al., 1991a]), or when a monolithic network has to learn two or more different tasks at the same time (*spatial crosstalk* [Jacobs, 1990]).

7. Reusability. Modular systems allow for the reuse of modules in different activities, without having to re-implement the function represented by each different task [De Jong et al., 2004, Garibay et al., 2004].

8. Reduction of the effects of the credit assignment problem. Whenever the controller must learn something new, the learning mechanism should provide a learning signal based on the controller's current performance. This learning signal will be used to modify the controller parameters for an improved controller behaviour. When the controller is in charge of many elements it becomes difficult to find which parameter has to be changed based on the global learning signal. Modularization helps to keep controllers small, minimizing the effect of the credit assignment.

3.1.4 Drawbacks of the modular approach

The drawbacks of modular systems are in general as follows:

1. In general, no systematic method exists to decide which and how many modules are required for the controller, nor which functionalities should

encode each one. This information is usually heuristically decided by the designer, based on his experience [Brooks, 1986, Mataric, 1992, Schrott and G., 1995, Mataric and Cliff, 1996, Vlassis, 2003].

2. Different modules working on different tasks inside the same robotic body implies designig a coordination system between modules.

3. In the particular case of modular neural networks, there exists the additional drawback of credit assigment in the sense that neural modules need to be trained, and it is not clear how to teach every module what to do.

3.2 A taxonomy of modular neural architectures

This section provides a classification of modular structures, - especially those based on artificial neural networks, since the purpose of this thesis is the use of ANNs for robot control. It will help to identify interesting approaches for the application of modularity to the evolution of complex controllers; an already existing architecture, a modified architecture, or a new designed one. The taxonomy introduced is based on the classification presented in [Buessler and Urban, 2003] on modular neural architectures. A similar classification can also be found in [Auda and Kamel, 1999], including a classification of modular training algorithms. This section also contains a list of particular instances of those architectures when applied to robot control. All the architectures included will be considered to be homogeneous, that is, all modules within an architecture will have the same internal structure, except for the case of automatically generated architectures.

3.2.1 Hierarchical architectures

In hierarchical architectures (see figure 3.6), neural modules are implemented in a layered cascade of modules, with the main role of the structure being to determine which module will finally generate the answer to the task. In the cascade, output from previous modules select which module will be activated in the next hierarchical level. The system's final response is provided by the last module activated, better known as the output module. Supervised learning techniques are usually used to train output modules, while unsupervised learning is applied to the selection modules. This architecture is primarily used in task recognition patterns, [Murino, 1998], characters [Suganthan, 1999] or vowels [De Sa and Balard, 1998].

In robot control research, unsupervised and reinforcement learning-based methods have been used mainly for training modular hierarchical architectures. One interesting application is [Nolfi and Tani, 1999], and revolves around the use of *prediction learning* to extract *regularities*[2] from the external environ-

[2]The term regularity is understood as a set of sensory states which can be separated from others and which are stable over a period of time, hence predictable. The extraction of regularities from the environment is meant to be useful for the achievement of the goal.

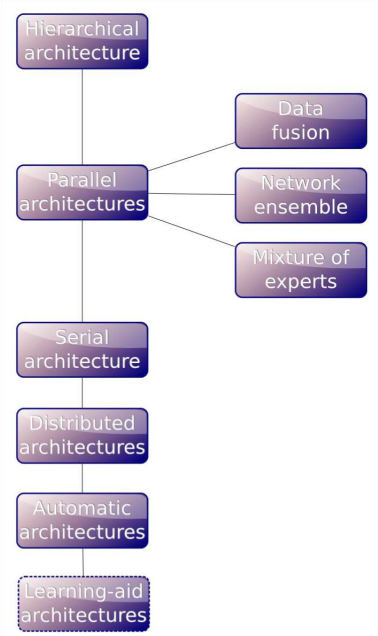


Figure 3.5: General classification of modular neural architectures (adapted from [Buessler and Urban, 2003]).

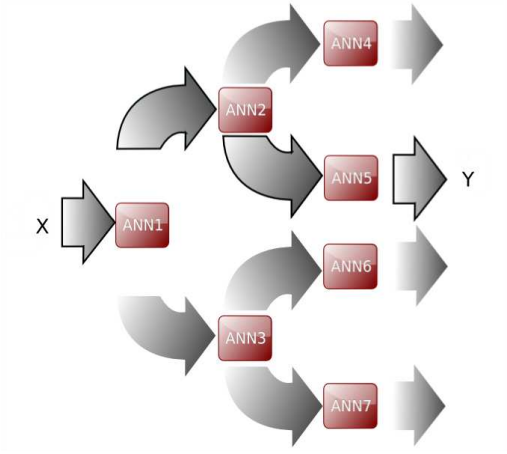


Figure 3.6: Hierarchical architecture. Thick black arrows show the path followed by input data X to generate answer Y.

ment. A similar approach was used in [Schmidhuber and Prelinger, 1993] for an unsupervised classification.

In a more recent related work [Paine and Tani, 2004, Paine and Tani, 2005], research focused on how hierarchical controllers can self-organize without explicitly designing that hierarchy.

In a different line of research, modular architectures decomposing the control problem into a set of neural modules exist, which should interact for the resolution of the robot task, in this way generating a hierarchy of modules. This is the case of the *Emergent Task Decomposition Network* (ETDN) in [Thangavelautham and D'Eleuterio, 2004].

3.2.2 Parallel architectures

In parallel architectures several modules treat input information in parallel, and modules can use either all the same information, or different subsets of it. Generally speaking, a problem is decomposed into several simpler sub-tasks, each one managed by a separate neural net (a module). Each module's solution is then integrated using an integrator module, thereby generating the system's final answer. Parallel architectures have largely been used for classification problems since they are suitable for an easy partitioning of the input space into the different categories.

Several types of parallel architectures exist depending on how input information is introduced, and how the final output is generated. In the case that each module treats its own and different input information the architecture is referred to as *data fusion architecture*. This type of architecture is composed of several modules, each one having a different set of inputs. An example of this is the sensed data coming from different sensors, whose responses are combined in order to generate the system's final answer (figure 3.7). The purpose of this architecture is essentially the combination of different data coming from different systems, along with the generation of a single percept. An application example can be found in [Ghahramani, 1995] where a sensorimotor integration system is constructed for localization using visual and auditory stimulus.

Another kind of parallel architecture is known as the *ensemble-based approach* [Hansen and Salamon, 1990, Krogh and Vedelsby, 1995]. In this case, all the modules have the same input data, and different outputs are generated. The results from modules are then either combined (known as (*cooperative* use), or only one is selected (known as *selective* use) by an additional module. The purpose of having different modules in the same data is to enhance the systems performance, rather than to divide the problem into subtasks [Alpaydin, 1993, Battiti and Colla, 1994, Rogova, 1994, Jacobs, 1995, Sharkey, 1996, Sharkey, 1997] (see figure 3.8).

The ensemble approach has generally been used in classification applications [Yao and Liu, 1996, Sharkey and Sharkey, 1997]. In [Pardoe et al., 2005] the ensemble approach was used to control a pole-balancing action.

Without a doubt the most popular type of parallel architecture is the *local experts ensemble*. In this case each module specializes in a portion of the

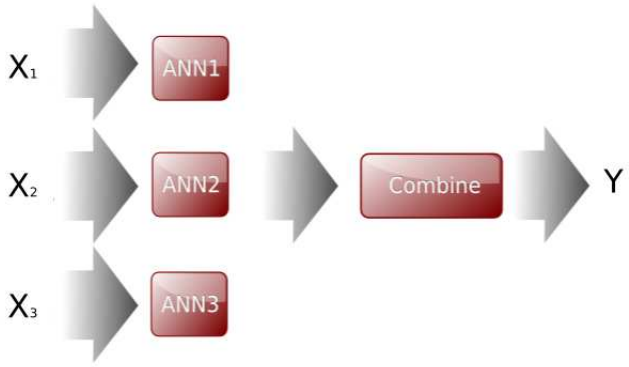


Figure 3.7: Data fusion architecture. Different information is feed into the modules.

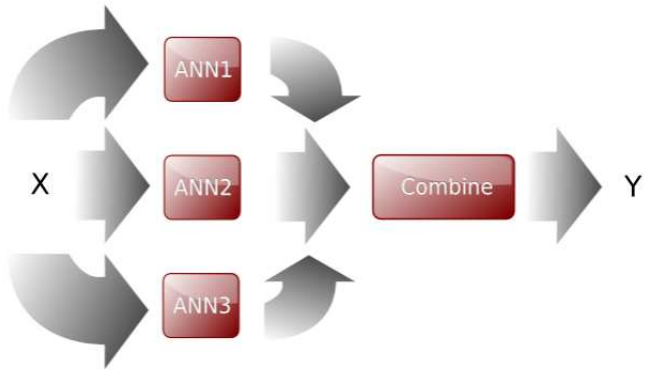


Figure 3.8: Neural network ensemble. The same information is provided to all the modules.

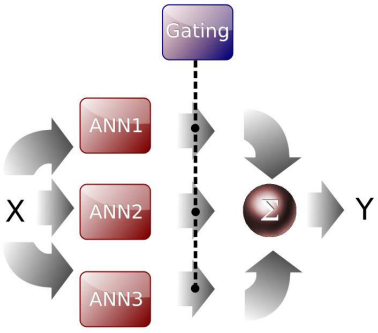


Figure 3.9: Mixture of experts.

search space being trained only with the part of the training set that corresponds to that space. Each module becomes an expert on that space, and the system's final answer is a combination of the answers from each module. This architecture has had more successful instances in the *adaptive mixture of experts* architecture developed in [Jacobs et al., 1991b]. Based on previous works in [Hampshire and Waibel, 1989, Jacobs et al., 1991a], a local experts ensemble with an additional gating network added was created which determined the weighting coefficients for each module output according to the input values (see figure 3.9) [Alpaydin and Jordan, 1996].

The same authors implemented a hierarchical version known as the *hierarchical mixture of experts* [Jordan and Jacobs, 1994]. A *modified hierarchical mixture of experts* architecture was proposed in [Azam, 2000], which is a combination of the original architecture and the alternate hierarchical mixture of [Xu et al., 1995] with two added elements; namely, input gates and input switching.

In the case of robot control, [Tani and Nolfi, 1999] studied the case of when neural networks self-organize and coordinate as a set of experts which will handle the different sensorimotor flow that a robot experiences.

In [Chen and Chi, 1999], what the authors refer to as a *generalized mixture-of-expert* architecture was implemented. The main idea of the basic architecture was kept, but groups of expert ensembles rather than simple expert neural nets were created. The learning method was also modified by using their own probabilistic method.

In the same line, [Auda and Kamel, 1997a] defined the *cooperative modular neural network* architecture, which is a modified version of the mixture of experts where the gate is substituted by a voting mechanism implemented in each of the expert modules. They provide an extensive comparison between their architecture and others for classification tasks [Auda and Kamel, 1997b], including some parallel architectures such as *Decoupled* and *Other-output* architectures [de Bollivier et al., 1991], *ART-BP* [Tsai et al., 1994], *Ensemble with majority vote* [Alpaydin, 1993], *Ensemble with average vote* [Battiti and Colla, 1994],

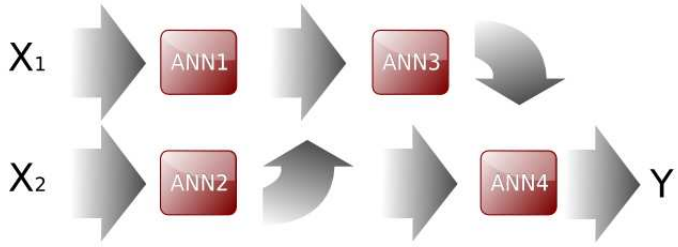


Figure 3.10: Schematics of a serial architecture implementation example

Merge-glue [Hackbarth and Mantel, 1991] and some hierarchical ones like *Hierarchical* architecture [Corwin et al., 1994] and *Hierarchical Competitive Neural Net* [Auda, 1996].

In a different work, [Davis, 1996] created a modular neural network for the autonomous navigation of robots. In [Yamaguchi and Itakura, 1999] a similar modular neural architecture with a different learning algorithm based on feedback error training was proposed. Another case is that of [Silva and Ribeira, 2003] where a modular neural network was used for navigation in a NOMAD 200 mobile robot.

3.2.3 Serial architectures

In serial architectures, complex problems are split into successive partial tasks (figure 3.10). Serial architectures are composed of several modules, each one handling a few inputs and producing an intermediate result that is fed into another module until the final module produces the system answer. They can be trained by supervised learning by making use of the backpropogation error technique; using the final error obtained by the final module as the error signal, and backpropagating it through the different modules. However, in some cases with multiple levels of serial modules this method can be difficult to implement. In such cases a separated training by modules could be used, - though it is not assured to converge on an intelligent controller for all applications.

In [Caelli et al., 1999] this architecture was applied in a pattern recognition task. For robot control, [Buessler and Urban, 2003] applied the serial architecture to the control of a robot arm in combination with a learning-aid architecture to train modules (see subsection 3.2.5). The learning technique, known as *bi-directional learning*, and consists of the addition of a learning-aid module which trains itself simultaneously to the module that is supposed to be trained by tracking each other's response as the desired output to minimize the difference between their estimations.

Another training option is the use of genetic algorithms to evolve some of the modules. This is the case of the legged robot of [Manoonpong et al., 2005,

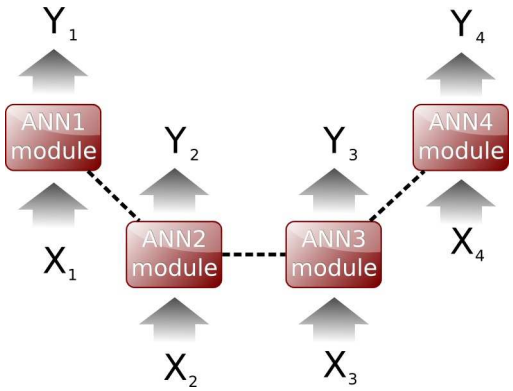


Figure 3.11: Distributed architecture schematics. Each module manages a small part of the control, and can be linked by a coordination mechanism (dotted line) with the other modules.

Manoonpong et al., 2007] controlled by neural modules. A series of three different modules was designed: one in charge of accessing the sensors and preprocessing them, another to generate the oscillatory patterns, and a third to control the velocity. Each module was successfully trained using a genetic algorithm.

3.2.4 Distributed architectures

Distributed architectures are those without a central controller. There is no module to combine the modules' answers or to select which module will produce the control response. Instead, all the modules produce their response in the effectors that they are associated to, and their response is usually *coordinated* in some way with the responses of the other modules. This means that the output produced by each module is not entirely independent of the output of the other modules, but rather that a subtle influence exists. This is a very important difference when compared with the previously described architectures, where each module produced its own answer in a fully isolated manner.

In distributed systems, coordination can be achieved through communication between elements [Werner and Dyer, 1992], though communication is not strictly necessary to achieve coordination. Some studies in fact indicate that communication may even be harmful when a system remains simple as it can overload the system, resulting in a sub-optimal solution [Wagner, 2000]. In [Yong and Miikkulainen, 2001] for example, a distributed system that coordinates with communication is compared with another without communication (only the environment provides coordination). They concluded that communication does not provide an advantage when systems are very simple.

Coordination in some systems can therefore be obtained by other means; depending on the nature of the task to solve, environment, and available pop-

ulation, etc (like for example simple rules [Kaplan, 2005]). An example of this approach is [Wischmann et al., 2005], where a decentralized controller for the control of a very special robot with a wheel shape was evolved. In this case, a wheeled robot was controlled by means of five different and entirely independent controllers. Coordination is obtained then by an interaction of all the modules contained in the agent, which acts on the environment [Pfeifer and Scheier, 1999].

In [Potter and Jong, 2000] a similar method for the distribution of components by the cooperative coevolution of subcomponents was described. The evolutionary process provides the required pressure for interdependent subcomponents to emerge, - each one covering one portion of the solution space. Each subcomponent is then able to adapt to changes in role of the other subcomponents. Their job was unfortunately only applied to pattern matching problems.

Distributed approaches have been especially successful in the generation of walking controllers. Even if most references propose ad-hoc solutions (that is, solutions without generalization for use in other robots), they are effective in what they attempt to solve. The first examples are [Beer and Gallagher, 1992, Gallagher et al., 1996], where a series of distributed neural modules is used to simulate an insect walking. Each module controls one of the legs, and a coupling between modules is used to coordinate the different modules.

A similar approach was used by [Ijspeert, 1998, Ijspeert, 2001], where distributed components were used for the generation of a walking and swimming salamander. In this work, separated neural modules were evolved, each one implementing an oscillator, and designing one of these modules for each segment of the salamander model. Couplings between different segments were then evolved, obtaining a complete coordinated distributed controller. The same idea of coupled oscillator modules was also applied to the evolution of swimming lampreys [Hallam and Ijspeert, 2003].

Recently, [Valsalam and Miikkulainen, 2009] has proposed a method to automatically discover symmetries in the task to evolve while evolving the distributed controller required for it. They applied the method to the evolution of a walking quadruped in simulation.

Additional examples of distributed architectures applied to legged robots can be found in [Lewis et al., 1992, Valsalam and Miikkulainen, 2008].

3.2.5 Learning-aid architecture

In monolithic networks supervised training is easily achieved by using simple algorithms (like backpropagation [Bishop, 1995]). In modular architectures this is not possible though, - at least not in a direct way, as error produced during the training phase is only accessible at the global controller level. Instead, learning should be defined at the level of each individual neural module to train each of the modules.

In [Buessler and Urban, 2003] an architecture known as *learning-aid* was considered in an effort to solve this problem. Learning-aid modules in the architecture were used to train neural modules during the training phase. Those learning-aid modules were also made up of neural networks, but their work was

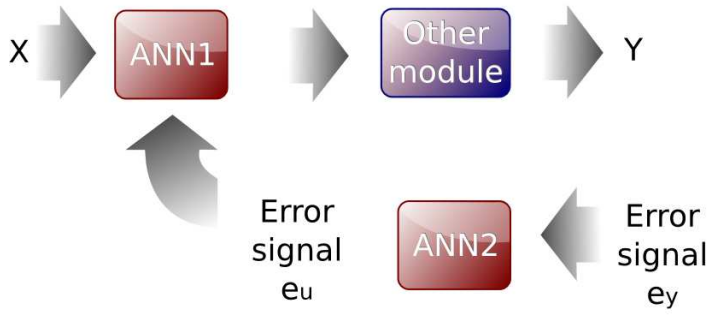


Figure 3.12: Learning-aid architecture schematics

reduced to the training phase. Once the training phase was over those modules were not used for the generation of the controller response. Figure 3.12 presents a basic schematics of this architecture, where the ANN2 neural module is included to provide an error signal to the ANN1 module, thus allowing it to learn.

Unfortunately, not much research exists on how the ANN2 module should be created, nor which type of signal it should receive in order to generate the learning error for the ANN1 module. An application of this architecture can also be found in [Buessler and Urban, 2003]. A serial modular architecture was coupled with a learning-aid architecture, which allowed the supervised training of the modules; avoiding the separated training of each module [Buessler et al., 2002, Buessler and Urban, 2003, Hermann and Urban, 2003].

Another application can be found in [Yamaguchi and Itakura, 1999], where a robot neural architecture is divided into submodules, and each of these trained by simple feedback controllers.

3.2.6 Automatically generated neural modular architectures

In previous architectures, schematics and distribution of modules were designed and classified by human designers following particular engineering guidelines. Another type of architecture, defined as *automatically generated* is now defined, inclusive of all the methods that automatically generate the neural structure of a modular controller. The methods described here do not present a predetermined distribution of modules, but rather an algorithmic description of how to generate them. Furthermore, modules created following this procedure may not even be homogeneous since they are created through a genetic algorithm, making it impossible to define beforehand the type of architecture that will appear.

Thus, it is very likely that architectures generated using these methods could

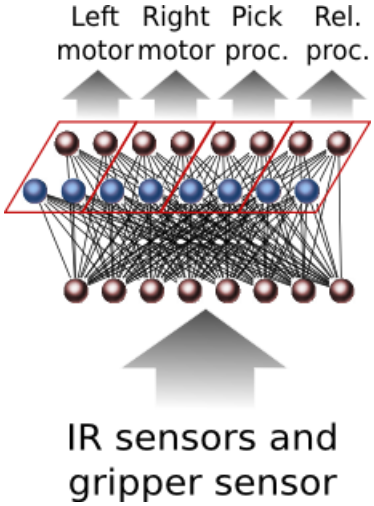


Figure 3.13: The emergent modular architecture in [Nolfi, 1997]

be classified within the previous types of architectures (serial, parallel, etc), but we have considered this category an additional one as it is not limited to the generation of already existent architectures, but rather that any mixture and combination can also be generated, as long as the final controller is composed of modules.

Basically, automatic architectures are generated through the execution of a genetic process which decides some or all of the following requirements: how to split search space, how many modules are required, how are they implemented, and how they should be combined for the generation of a solution. Application examples can be found in [Sims, 1994, Gruau, 1995, Whitley et al., 1995, Bongard, 2002, Garibay et al., 2004, Reisinger, 2003, Reisinger et al., 2004].

In [Nolfi, 1997], an unsupervised modular neural architecture, referred to as *emergent modular* was proposed for the control of real robots (see figure 3.13). Based on this approach, architectures for the automatic creation of neural modules were presented in [Calabretta et al., 1998, R. Calabretta and Wagner, 1998, Calabretta et al., 2000].

[Di Ferdinando and Parisi, 2000] studied the use of modular neural networks to solve the neural interference problem, using [Rueckl et al., 1989] as point of departure in the *What and Where* task. This research was extended to an appropriate modular and innate connectionism method based on Artificial Life, called *evolutionary connectionism* [Calabretta and Parisi, 2005]. Research related to this problem was continued in [Calabretta et al., 2003].

In [Watson, 2002a, Watson, 2002b], *compositional methods* were proposed as a solution to the generation of complex behaviours with high interaction between them. They basically consist of a set of evolutionary operators with enhanced functions, such as sexual recombination with several lineages in di-

vided populations, lateral transfer and symbiogenesis. A description of how to use them for the resolution of numerical problems exists, but no application to robots has to date been reported .

There also exist examples of automatically generated controllers having their own specific methodology. An application example is found in [Liu and Yao, 1997], where individual modules and their integration are evolved in the same evolutionary process. Neither the number of modules nor the integration fixed by any algorithm was predefined. This approach avoids the separation of module design and module integration, - as adopted by other solutions. This method was applied to the diagnosis of several diseases.

In a similar work, [Cho and Shimohara, 1997] studied how structure and functionality of modular neural networks emerge by designing some basic *modules*. Modules were the basic building blocks used to autonomously evolve the structure of neural networks, developing at the same time a new functionality. It has been applied to a handwriting recognition system.

In [Stanley and Miikkulainen, 2002], the *NEAT algorithm* has been proposed for robot control where an ensemble of neural nets are used for the task. Examples of application include [Pardoe et al., 2005], where the NEAT algorithm evolved at the same time as the architecture of the nets, their weights and the gating network. The NEAT algorithm has lately been improved to Modular NEAT [Reisinger et al., 2004], which has the ability to evolve reusable neural modules.

In other cases, genetic algorithms are used to evolve only small parts of the controller modules. For instance, in [Filliat et al., 1999] a 6 legged robot walking controller was designed using an evolutionary process that automatically evolved two different modules; one for walking and another to avoid obstacles. The algorithm evolved the inner workings of the modules, but the number of modules, their functions and how they connected to each other was decided beforehand.

3.3 Selecting a suitable architecture

Thus far a list of the currently available types of modular architectures for neural network control has been presented. By comparing them it can be observed that apart from distributed and automatic architectures the process to generate a modular controller is very similar. Simply stated, it starts by dividing the task into modules, next, decomposition into modules is performed by deciding which sub-tasks or behaviours are required to generate the global task, and finally, a module is assigned to implement each of such sub-tasks [Auda and Kamel, 1997b, Azam, 2000].

This kind of decomposition into sub-tasks works well in simple robots with a limited number of devices, but this type of decomposition can not lead to an evolved controller when the number of devices is high, such as in the case of complex robots. Applying this approach to complex robots is not as simple since the number of devices required to implement the behaviour will be large

even if the behaviour is simple.

Hierarchical architectures for instance require that all the sensors feed their sensed values into a single input module, generating the actuators commands from another single output module. When the number of devices is high, a single input module and a single output module do not seem to be enough, as spatial crosstalk can appear to be preventing training. A possible solution is to divide input and/or output modules into smaller modules; each one in charge of a reduced group of devices. Since it is difficult to decide which devices are controlled by which module, a more general solution would be to assign a single module to each device. Hence, no single input module would exist but a group n of input modules equal to the number of sensors. For the output the same applies. Output would also be composed of p different modules; each equal to the number of actuators. This solution, even if it looks more adaptable, still faces the challenge of training all of those small modules and their internal logic modules, which appears quite difficult due to the large number of modules and their relations.

If the idea of one module associated to each device is kept, but all the internal logic removed, the hierarchical architecture results in a kind of parallel one, composed of one input module per sensor, and where the combination module is composed of several modules, - one per each output actuator.

Both parallel and serial architectures face the same problems as hierarchical ones when applied to complex robots. Many sensors and actuators need to be managed, which in turn makes the number of connections in the input/output modules very high. This complicates the training of the intermediate modules to a large extent (gating, combination or intermediate networks).

Distributed architectures and automatic architectures usually have different approaches. Automatically generated architectures could appear useful at first glance, but they include a large search space when they are applied to complex robots. Since everything needs to be evolved from scratch (modular structure and network weights), problems for evolutionary robotics increase as the robot becomes increasingly complex. From those methods however, one can borrow the idea of evolving inner parts of modules and connections between them, but only once a suitable architecture is already in place and being used as substrate.

In distributed architectures, a single module can be in charge of controlling a small group of devices whilst implementing a simple function. For instance, in walking architectures shown in section 3.2.4, small neural modules implemented oscillators to act upon a single actuator. This mechanism allows a considerable reduction of the number of inputs and outputs that the neural module has to train, simplifying the task to be implemented by the module as well as the number of weights to evolve. This type of approach can also benefit from symmetries or similar functionalities between elements by evolving one single module and then duplicating it whenever the same functionality is required inside the robot (as was done in [Ijspeert, 1998, Ijspeert, 2001, Hallam and Ijspeert, 2003]). This reuse of modules is also very efficient at reducing the evolutionary space of the whole robot. A drawback to date however is that distributed approaches have only implemented ad-hoc solutions to the robot and task to be solved at the

moment. No general procedure exists to apply to any robot for any task.

For the resolution of the problem depicted in chapter 1, a modular architecture which reduces complexity there where it is is required. Then, in complex robots, the first source of complexity is the number of devices to control for any task. Any simple task will have to manage a high number of devices to control, hence, the evolutionary search space will be very large for even the simplest of tasks.

This is more in the line of a distributed architecture, where modules are created based on the number of the devices to control (indicating the complexity of the robot), instead of modularizing tasks. For this purpose, the introduction of a new way of performing modularity is proposed, which can in turn lead to more general and scalable modular controllers. Instead of performing modularization at the task level, that is, a global task is divided into sub-tasks, we propose to perform a modularization at the device level, that is, a global controller is divided into sub-controllers; one per each device. Modularity in terms of sub-tasks is not forgotten. A complex robot can also be required to implement quite complex tasks. For this reason, modularization at the level of the task level will also be required.

We will refer to this approach to modularization as *strategic* and *tactical modularity*, and a complete description is provided in the following chapter.

3.4 Conclusions

This chapter has provided a complete review of the different types of existing modular neural networks; focussing on their application to robot control. It has been divided into two main approaches: modularization based on task decomposition (mainly implemented by hierarchical, parallel and serial architectures), and modularization based on the devices that are involved in a task (in the case of distributed architecture). It was observed that for complex robots, modularity has to be implemented first where complexity is found first, that is, in the number of devices. The next chapter will provide an in-depth description of our proposed architecture.

Nothing is more fairly distributed than common sense: no one thinks he needs more of it than he already has.

Descartes

4

DAIR, a distributed modular neural architecture

In the previous chapter, a complete review of modular neural architectures that have been successfully applied to robot control has been provided. However, none of the modular approaches presented were applied to complex robots with dozens of devices to be coordinated.

Distributed architectures appeared to be the most suitably matched to the requirements initially established in this thesis; namely, applicable to complex robots, general enough to be applied to any robot and any task, scalable as to the number of robot devices (sensors or actuators) as well as in the number of tasks, able to generate control for an agent with the introduction of the least amount of external knowledge as possible, and that easy integration into the controller should be possible when external knowledge is required .

This chapter is devoted to detailing the architecture developed by this research to address those precise requirements. The ideas that drove the development of the architecture will be discussed, as well as their influence in the final version. Finally, several illustrative application examples of both simple and complex robots will be given.

4.1 Departing ideas

As a result of the analysis conducted in the previous chapter, we have established that a distributed modular architecture is required for the purpose of this thesis. Little is known however about at what level of modularity should be implemented (how to generate the modules), how modules will be coordinated, and how modular elements should be encoded and trained. This section provides a list of important ideas on the generation of intelligent agents that have been taken into account when designing the proposed architecture.

4.1.1 Sensorimotor coordination

Sensorimotor coordination will be the basis of the architecture's design for the generation of complex robot behaviours. Sensorimotor coordination is un-

derstood as the coupling existing between perception and action when performing a behaviour. The system will integrate multiple sources of information in order to estimate its own state, and its relation with the environment [Ghahramani et al., 1997].

In fact, the *principle of sensorimotor coordination* [Pfeifer and Scheier, 1999] will be adhered to; which states that any intelligent behaviour, such as perception, categorization, memory, etc, can be conceived as a sensorimotor coordination which serves to structure the input. As shown in [Pfeifer and Scheier, 1997], sensorimotor coordination does not simply mean reflexive behaviour. Instead, sensor inputs and interaction with the agent environment whilst performing a task serve to guide the behaviour.

The concept of sensorimotor coordination is particularly interesting for three reasons:

- Both sensors and motors play an important role in coordination, and should therefore have an equivalent level of complexity in control. Both should co-evolve to obtain coordination when the robot is complex.
- Sensorimotor coordination induces correlations, thereby reducing the high dimensional space to a lower dimensional sub-space.
- It allows for the integration of several sensory modalities.

The sensorimotor coordination principle suggests that the whole process of behaviour does not begin with a sensory stimulus, but rather with a sensorimotor coordination, and that it is the movement which is primary, and the sensation which is secondary (as indicated in [Pfeifer and Scheier, 1997] quoting [Dewey, 1896]). Sensor and motor systems must be intimately connected in their behavioural execution. This principle will serve as the basis for both the control architecture and design in this thesis. Additionally, as will shown in chapter 8, this principle will permit the architecture to construct its own internal representation of its current situation, directly grounding experiences to its sensorimotor apparatus.

4.1.2 Massive modularity

Most modular approaches advocate for a, let's say, *discrete* level of modularity. Modularity for an agent's control system consists of a limited number of modules; each one in charge of controlling a simple behaviour, and a coordination mechanism which decides how they should be combined.

Modularity, however, can be pushed further and introduced into the behaviour modules. We advocate for the concept exposed in section 3.1.2.2 of *massive modularity* [Minsky, 1988, Carruthers, 2004, Carruthers, 2005] to break down large behaviour modules into smaller modular parts. Several levels of modularity are defined within the massive modularity of mind theory (the idea that the mind is composed of independent, closed, domain-specific processing modules). Even if the existence of modules at a functional level is recognized,

supporters of this theory also suggest the existence of modularity within lower levels of the brain. These modules would only be concerned with performing the simplest of tasks. When several of these small modules are put together - whether competing or cooperating with each other, the emergence of a *superior* function would result by means of a coordinated and concerted interaction amongst all the elements, and without a central control driving either the modules or the interaction. This is called *distributed artificial intelligence* [O'Hare and Jennings, 1996], which has derived into *multi-agent systems* [Stone, 2000].

We have adopted this idea for the design of the DAIR architecture, and use it to decompose large behavioural modules into smaller ones. This decomposition should permit the use of the architecture in complex robots with dozens of devices, in which it would otherwise have been very difficult to generate a behaviour.

4.1.3 Using human knowledge

It has been argued throughout this thesis that human knowledge needs to be introduced into the evolutionary process in order to obtain complex behaviours in a complex agent and successfully generate the agent controller. When comparing the artificial evolutionary process described in chapter 2 (see figure 2.1) with natural evolution, it has been observed that natural evolution gradually shaped the animals body plan, sensors and actuators, environment and nervous system *at the same time*. Artifical evolution however attempts to evolve the nervous system for a given body (the robot morphology), a group of given sensors and actuators, and a given environment (which includes the required task/behaviour in a given place). Only when the robot and the task to evolve have a low degree of complexity is the evolutionary process able to evolve the appropriate behaviour; this is due to the evolutionary steps between successful generations being very small. However, when the robot or the task is complex the evolutionary process cannot produce a good enough solution, and it falls into local minima or bootstraps that do not produce the required behaviour. In this case, only a gradual incremental change from one generation to the next may lead to a functional controller.

Intense research is currently underway in an attempt to develop better genetic algorithms and genetic encodings with which to decrease the required granularity for evolution, i.e. reduce the search space at each step. This thesis however proposes a different solution based on the introduction of human knowledge in the evolutionary process. This approach follows the line traced by incremental evolution (see section 2.4.1), where expert knowledge is used to design the different steps in which the evolutionary process will be divided. Similar suggestions have been proposed in other works; such as BAT methodology [Urzelai et al., 1998], or in the previously studied developments of controllers for different robots [Ijspeert, 2001, Hallam and Ijspeert, 2003, Muthuraman et al., 2003, Muthuraman, 2005]. Machine learning techniques (like the evolutionary process) are integrated in a rational manner in each case, along with a previous

in-depth analysis of the required behaviour for the agent. As was stated in [Urzelai et al., 1998], the entire development activity needs to be conceived and organized in the appropriate manner.

Accordingly, the DAIR architecture will be modularized for allowing the *straightforward* introduction of knowledge and providing as many opportunities as possible for their introduction. Furthermore, the knowledge introduced should affect only those parts of the controller that need to be affected, thereby avoiding the crosstalk problem whenever possible.

4.1.4 Double closure

We would like to consider to some extent the concept of *double closure*. The concept of double closure was described by von Foerster in his work on how to implement cognitive beings [von Foerster, 1970]. It basically states that:

> The meanings of the signals of the sensorium are determined by the motorium; and the meanings of the signals of the motorium are determined by the sensorium.

In the same way as stated in the sensorimotor principle, sensors and actuators must be coupled at a low level (sensations must affect motors, and motors must affect sensors; but in a sense that goes *beyond* the indirect resulting influence of the environment) in order to obtain an autonomous agent. It is this coupling between both systems that produces a meaningful representation of the world for the agent itself, and it is this internal model which should permit the agent to generate cognitive processes.

Accordingly, an embodied agent can be autonomous and organizationally closed, but at the same time structurally coupled with the environment it is situated in, which in the constructivist sense is its own invented reality [Ziemke, 2005]. It is evident that in most agent control systems, sensor processes and motor processes are separated even if feedback from the real world is included. This feedback is not enough to achieve the von Foerster concept of double closure. Perception and motion must therefore be connected to each other in such a form that information has its origin in this creative circle. Motor stimuli *must also be sent* to the sensor elements in order to allow a correct *prediction*. Hence, the control mechanism must not be based on internal models which try to precisely model the environment; instead, the internal model must induce the emergence of an invented reality based on interaction with the environment.

4.2 The DAIR approach

The most important issue when designing modular neural networks is the definition of the modular neural architecture at the abstract level such that it does not lose its relevancy to the application at hand, biological/cognitive adherence and theoretical analysis [Azam, 2000]. Hence, based on the definitions provided

Figure 4.1: Main steps in the design of modular neural networks (reproduced from [Auda and Kamel, 1999])

in section 4.1, a complete definition of all the relevant terms and parts that constitute the DAIR architecture is included in this section.

4.2.1 Introduction

As indicated in [Auda and Kamel, 1999], three general steps are common in most Modular Neural Network (MNN) designs: task decomposition, training and multimodule decision-making (see figure 4.1). Task decomposition refers to the division of the required behaviour into several sub-behaviours, and then assigning each sub-behaviour to a module. Task decomposition is the most important step when implementing a modular neural network design. Subsequently, modules should be trained in either parallel or different processes - following the sequence indicated by the modular design; and finally, when modules have been prepared, a multi-module decision-making strategy must be implemented to indicate how all those modules should interact to generate the global response that will in turn give rise to the required global behaviour.

When closely observing the modular approaches referred to in the previous chapter, it becomes clear that most of them more or less adhere to the previous three-step specification. From our point of view, these architectures perform modularization *at the behavioural level*, that is, the main goal behaviour required for the agent is divided into several sub-behaviours (in one way or another) and one neural module is in turn created for each sub-behaviour. This method has also been referred to as *functional decomposition* in [Davis, 1996]. One problem with this approach however is that when the agent to control is complex (composed of several sensors and actuators), then even a simple sub-behaviour

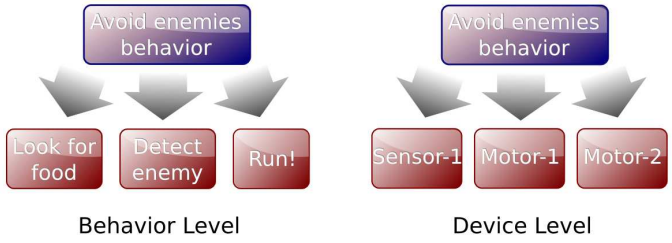


Figure 4.2: Differences between modularization at the behavioural level (left) and modularization at the device level (right), for a robot composed of 3 devices (1 sensor, 2 actuators).

will be difficult to generate as several of these devices need to be coordinated. Furthermore, it is possible that the given behaviour is an atomic unit (in terms of behaviours) and therefore cannot be divided into simpler sub-behaviours. A functional decomposition in behaviours results in a *nearly decomposable system* (as defined in section 3.1.1); where each behaviour is independent from the others.

Indeed, what may happen in such cases is that the evolutionary process is not able to find a good solution that coordinates all the robot devices as the search space is very large, due to the number of devices available (sensors and actuators). The following question then arises:

> If sensorimotor coordination is about coordination between sensors and actuators rather than coordination of different behaviours, why not use modularity there where the first source of complexity arises, that is, at the device level?

Reasoning about that question, this thesis proposes to introduce an additional level of modularization in the robot controller; performed at the *device level*. This modularization aims at providing a modularized structure within any behavioural module, but as well as the behavioural module being produced as a result of a functional decomposition, the device module is produced as a decomposition of resources; resources being understood as the devices (sensors and actuators) available for the robot to accomplish its assigned task (that is, its behaviour). This type of modularization leads to a *decomposable system* (as defined in section 3.1.1), where modules for each device are independent but continue to hold strong dependencies with the rest of modules.

Introducing modularization at the device level results into two different types of modularization. The next section defines both types and clearly determines at which level each one acts. Interaction between the two levels will also be described, as well as how they can be combined in order to generate extremely complex controllers.

4.2.2 Definitions

From game theory [McCain, 2003], it has been defined that *strategy* answers the question of what has to be done in a given situation in order to perform a given task, i.e. it divides the global target solution into all the sub-targets required to accomplish it. *Tactics*, on the other hand, answers the question of how the plans are to be implemented i.e. how to use the resources available at that moment to accomplish each of those sub-targets. When these definitions are applied to agent behaviour, strategy can be thought of as the overall group of behaviours (sub-goals) required by an agent for the accomplishment of a goal, and tactics as the actual means used to achieve each of those sub-goals. Thus, these same definitions can be used to identify two levels of modularity in neural controllers: *strategic modularity* and *tactical modularity*.

We define *strategic modularity* in neural controllers as the modular approach that identifies which sub-behaviours are required for an agent in order to obtain a global behaviour. This division into sub-behaviours can be made from a distal or proximal point of view, but either way, each sub-behaviour identified will be implemented by a monolithic neural net. In contrast, we define *tactical modularity* in neural controllers as the approach that identifies which robot devices are necessary for the implementation of a given behaviour, and creates a single module for each device. In tactical modularity, modularization is performed at the level of the elements that are actually involved in the accomplishment of the task.

To our knowledge, most of the research is based on neural modularity and divide-and-conquer principles, focussing on their division at the strategic level, that is, how to divide the global behaviour into its sub-behaviours; either from a distal or from a proximal point of view, and in either a manual or an automatic way. They then implement each of those sub-behaviours by means of a single neural controller. Our research proposes the use of tactical modularity, where a behaviour must be automatically broken down into the parts that are going to actually physically perform the work; namely the agent's sensors and actuators. It is expected that tactical modularization will be very helpful in the generation of controllers for complex robots, that is, robots where several sensors and actuators need to be coordinated in order to accomplish a task. In such robots, a monolithic implementation of the controller may not be possible even for a simple behaviour, and a modularization at the device level may quite possibly help to solve the problem. Additionally, early results (described in section 4.2.6) suggest that a greater level of modularity in the controller would increase its performance [Téllez and Angulo, 2004] of even simple behaviours in simple robots. This result will be confirmed in the next chapter, where the use of the two types of modularity will be compared against monolithic approaches.

The creation of modularity at the device level corresponds to four motivations

1. To design modules through the implementation of sensorimotor coordination.

2. To increase the level of modularity beyond the established models of sub-behaviours, thereby permitting massive modularity in robot control.

3. To allow information insertion by the designer directed at one particular device or small group of devices.

4. It would be a sound assumption to provide each element that participates in a task with its own processing element.

The use of one type of modularity does not prevent, in principle, the use of another one at the same time. Strategic and tactical modularity can be used separately or in conjunction with each other. When the robot task is not very complex either a strategic or a tactical modularization can in principle generate the required controller. In such cases we suggest that modularization selection depends primarily on the complexity of the agent, that is, a monolithic controller for simple robots or a tactically modular controller in the case of a complex one. Furthermore, when the task to solve is very complex and requires very different and unrelated behaviours, and the agent is also complex in terms of the number of sensors and actuators, then a combination of strategic and tactical modularization may be required. In such cases a strategic modularization should be performed first; this will identify the different sub-behaviours that require implementation. Following this, a tactical modularization should be developed for each of those sub-behaviours, identifying each of the devices that will participate in the sub-behaviour and assigning a module to each device. For instance, if an Aibo robot has to learn a behaviour to stand up, walk to some place, and then sit down, then a manual and distal strategic division of the task could distinguish between three different sub-behaviours (stand up, walk and sit down). Each of those behaviours would then be created using tactical modules (see figure 4.7).

We refer to the approach as *Distributed Architecture with Internal Representation* (DAIR), where both strategic and tactical modularity can be used in any combination. The term *Distributed Architecture* refers to the way different modules compose the architecture whilst no central control is in place. The addition of the term *Internal Representation* in the name of the architecture will be explained in chapter 8.

4.2.3 Implementing strategic modularity

The term *strategic modularity implementation* will be understood as the technique that identifies which sub-behaviours compose a given robot behaviour and also defines their relations in terms of cooperation. Strategic modularity can be implemented by any of the modular approaches referred to in chapter 3. How to perform the behavioural subdivision in terms of sub-behaviours has been widely studied in literature, and is not the goal of this research. In principle, any of the modularization methods described in chapter 3 are valid for integration with tactical modularity.

Examples of applied strategic modularity can be found in [Gruau, 1995], where cellular encoding is used to automatically decompose a problem into sub-problems, and then generate a single monolithic ANN for the solution of each sub-problem. Similarly, in [Thangavelautham and D'Eleuterio, 2004] an emergent task decomposition network architecture is proposed, consisting of a set of decision neurons which mediate competition, and a set of monolithic expert networks that compete for dominance. In a less automatic approach, in [Urzelai et al., 1998] the BAT methodology is proposed for a step-by-step design of modular controllers, with human intervention in the specification of the modules required. Developed in [Stone and Veloso, 2000] and lately improved in [Whiteson and Stone, 2003] a task decomposition scheme is proposed; namely *layered learning*, where a hierarchical task decomposition is manually performed, and learning applied to each separated layer.

In conclusion, strategic modularity has already been used for a number of years, - although it was not given that name. We have used the term *strategic* for these modular approaches in order to differentiate them from the new level of modularity that we propose; namely *tactical modularity*.

4.2.4 Implementing tactical modularity

Tactical modularity creates modularity at the level of the robotic devices which have to perform a required behaviour. Each of the sensors and actuators that participate in the strategic module conform to a tactical module. Tactical modularity is implemented by designing a completely distributed control architecture composed of small processing modules around each of the robot's sensors and actuators. Requirements for a tactical module are:

- It must control its associated device.
- It must have a single processing element, which permits its associated device to perform a simple task.
- It must be connected to the other tactical modules for coordination, since there is no central referee; especially important is the connection between sensors and actuators for the accomplishement of the double closure principle.
- It must be simple.
- In order to have homogenity, it must be possible to use the same module for either sensors or actuators.

Following these requirement set out, we propose the following design for tactical modules:

Each tactical module in the architecture is organized into what it is known as an *Intelligent Hardware Unit* (IHU). The IHU schematics are displayed in figure 4.3. An IHU is composed of a sensor or an actuator as well as a processing element that processes its associated device's information, that is, information

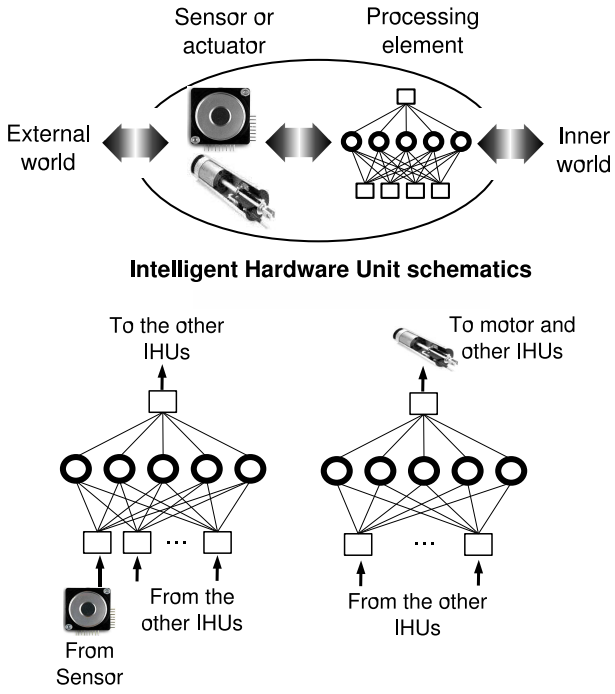


Figure 4.3: Top: Intelligent Hardware Unit schematics. Bottom: Internal connections of a IHU with a sensor (left) and an actuator (right).

received from the sensors or commands sent to the actuators. This type of connectivity means that the processing element is the one that decides which commands must be sent to the actuator, or how a value received from a sensor must be interpreted. We say that the processing element is *responsible* for its associated sensor/actuator.

All the IHU's are connected to each other, that is, output from each IHU is sent back to all the other IHU's, and each IHU is therefore aware of what the others are doing. The processing element is therefore also responsible for deciding what to communicate to the other elements, as well as interpreting what the others are communicating. IHU output take its own and the other elements information into consideration when generating its answer. This situation allows for a strong coupling between devices, as well as a response generation that is highly influenced by the other elements. A sensor IHU may generate a different answer for the same sensory stimulus, if information from the other IHU's are different.

Artificial neural networks have been selected as the processing elements for IHU's, since we are interested in modular neural networks. However, any type of processing element can be used (look-up table, regression, dynamic equations, fuzzy-logic, bayesian rules, etc). ANN type is not restricted either. In principle, we would like to use the simplest possible type for the task to be solved; this to be determined by the designer at time of implementation. In any case, each neural network will be endowed with as many inputs as IHU's exist, since all of them should be interconnected. As far as outputs go, only one will be allowed, and it alone will represent the output of the IHU module. Two or more outputs being generated would however allow for more complex models. In this thesis we have selected the single output option in order to reduce the overall complexity. Figure 4.4 shows a connectivity example in the generation of a DAIR neural controller in a simple robotic system composed of two sensors and two actuators. The processing elements are all distributed throughout the robot body, and strong interactions exist between the different elements. This means that even if the controller is created by separated modules, they are not really independent in the sense that the optimization of one module depends on the optimization of the others. Hence, this modularization can be defined as decomposable (as defined in section 3.1.1).

It should be noted that when several IHU's are put together in a control task, that each element has its own particular vision of the situation; selecting an action for its device based on its knowledge of the global situation at that moment as well as the state of its particular device. A distributed coordination between all the elements is required in order to permit the entire robot to perform the required behaviour; only possible due to the connections between all the elements. Coordination is achieved during the evolutionary process through the evolution of the weights of the inputs that connect to the outputs of the other devices.

What is interesting about sensors is that their associated IHU computes what to do with the sensed value. It determines whether the processing is necessary and then of which kind; depending on a number of factors: what

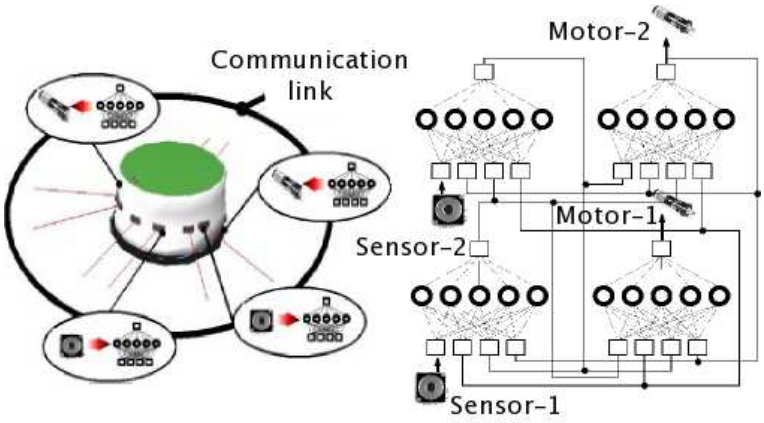


Figure 4.4: Simple application of the architecture for a Khepera II robot using two sensors and two actuators. The figure on the left displays a representation of the idea applied to the robot, and the figure on the right shows how such controller would finally look when implemented using the DAIR architecture.

other sub-agents are doing, the type of neural net used, the sensor features and the current value sensed. Hence, sensor processing is also learnt during co-evolution.

Distributed coordination

Distributed coordination can be defined as the result of a dynamical process that enables several agents to coordinate in order to achieve a global task without the use of a central coordinator [Kaplan, 2005]. The coordinated behaviour of the group emerges from the interaction of these simple agent behaviours. The concept of distributed coordination has been applied to several domains, e.g. the autonomous generation of a language for a group of agents [Kaplan, 2005], the generation of a walking agent, or in predator and prey games [Yong and Miikkulainen, 2001], to name but a few.

In the DAIR architecture, distributed coordination is required in order to achieve the coordinated behaviour of all the parts of the robot where an ANN for each sensor/actuator exists. Coordination will be possible because there is a direct *communication link* between all the IHU elements which reports their output to the rest of the IHU's. Given that different ANN's need to evolve through the evolutionary process for different roles in the common task, a co-evolutionary algorithm is required, i.e., the simultaneous evolution of several nets with a common fitness [Moriarty and Miikkulainen, 1998]. By using this kind of algorithm, it is possible to teach the networks how they must cooperate to achieve a common goal (i.e. the behaviour to be implemented) when each network has its own unique vision of the whole system.

Evolutionary algorithm

Being that the evolutionary process is a kind of reinforcement learning task, multiple algorithms can be used to generate the final controller. In fact, the selection of one algorithm or another can alter the degree of success achieved by evolutionary robotics methods [Nolfi and Floreano, 2000]. The DAIR architecture does not request the use of any specific algorithm, but simply recommends one that is particularly suitable for distributed architectures such as DAIR. After having researched the available algorithms (as described in section 2.3.1), we opted for Enforced Sub-Populations (ESP) [Gomez and Miikkulainen, 1996]. An ESP is an algorithm derived from the Symbiotic Adaptive Neuron-Evolution algorithm (SANE) [Moriarty and Miikkulainen, 1996a, Moriarty, 1997]. Both algorithms are characterized by their evolution of *neuron populations* instead of evolving complete neural nets. Populations consist of hidden neurons, each with its own input and output connections. Those neurons are used to construct the hidden layer of a neural network by randomly selecting neurons for the hidden layer. The difference between SANE and ESP is that in ESP the neuron population is segregated into sub-populations, allowing for a group of neurons to specialize in a specific position within the hidden layer of the neural network, hence obtaining ANN's with better fitness scores.

Furthermore, by keeping the same position it is possible for the ESP to evolve recurrent neural networks, since hidden neurons from the pool of neurons will always connect in the same way within a constructed neural network. The DAIR architecture does not specify the type of neural net attached to the devices within IHU's. As a general rule, the simplest possible option for the behaviour required should be used. However, as will be seen in chapter 6, complicated neural nets which include recurrent connections are sometimes required. For this purpose, ESP is a good option as an evolutionary algorithm.

Additionally, some reports suggest that ESP is especially good when used to evolve several neural nets belonging to different agents inside the same problem, but each with different vision of it [Yong and Miikkulainen, 2001]. This application of the ESP algorithm to multiple agents has been named *multiagent-ESP* (see figure 4.5).

A brief description of the ESP algorithm stages follows:

1. Initialization. The number of hidden units in the network, u, is specified and u subpopulations of neuron chromosomes are created. Each chromosome encodes the input and output connection weights of a neuron with a random string of real numbers.

2. Evaluation. A neuron is randomly selected from each subpopulation to form a network. The network is submitted to a trial in which it is evaluated on the task and rewarded with a fitness score. The score is added to the cumulative fitness $ftotal_i$ of each neuron i that participated in the network. This process is repeated until all the neurons have participated in an average) number of trials tav (by default $tav = 10$).

3. Recombination. The average fitness fav_i of each neuron is calculated by dividing its cumulative fitness $ftotal_i$ by the number of trials nt_i in which it actually participated ($fav_i = ftotal_i / nt_i$). For each subpopulation, neurons are ranked by their average fitness. Each neuron in the top quartile of its subpopulation is recombined with a higher ranked neuron in the same subpopulation using one point crossover and low probability mutation. Offspring are used to replace the lowest ranking half of the population.

4. Repetition. Repeat the evaluation and recombination processes until either the fitness reaches a certain score or the number of iterations reaches the limit; both previously set by the designer.

Additionally, ESP is combined with an iterative search technique, known as *Delta-Coding* [Whitley et al., 1991], which allows for a search for optimal modifications for the current best solution by fine tuning the solutions found. It works as follows: when the ESP algorithm has converged on a good enough solution (which means that diversity within the sub-populations is minimal), the best solution is saved, and all the rest discarded (deleted). A new group of subpopulations is then created by taking the best saved solution and adding to it the Δ-values representing small differences from the best solution - obtained from a Cauchy distribution. These sub-populations are used to start the evolutionary process again, now using the newly generated sub-populations. Delta-coding therefore performs an exploration of the solutions space in the vicinity of the best previous solution. The use of delta-coding facilitates incremental evolution.

Delta-coding is mainly used in ESP when incremental learning is performed. In such cases, delta-coding provides a good transition mechanism from one task of the incremental learning process to the next. Delta-coding is invoked when one of the tasks of the incremental learning process has been achieved, and the next task of the incremental learning process is then set for evolution. Subpopulations will look for a controller for the new task starting from a population of genotypes that lies around the solution of the previous task.

For the DAIR architecture the *multi-agent ESP* version of the algorithm is used. It is a modification of the basic ESP algorithm. ESP is usually used to evolve a single neural network. Due to the fact that we have several neural networks that must be co-evolved at the same time we can use an independent set of sub-populations for each network that we have to evolve, and evolve the complete set of sub-populations at the same time.

Figure 4.5 shows differences when using ESP or multi-agent ESP for the control of a robot composed of two motors and two sensors. In the case of *plain* ESP, a global ANN is evolved to control the whole robot, that is, sensors feed the network input and ANN outputs encode in some way what the agent must do at the next time step. In the case of multi-agent ESP several nets must be evolved, each one controlling a single part of the robot (one network for each device in the case of DAIR).

When several ANN's are evolved at the same time there are two different ways of assigning credit to them: using a *competitive approach* or using a *cooperative approach*. Since the DAIR architecture is a cooperative one, a cooperative

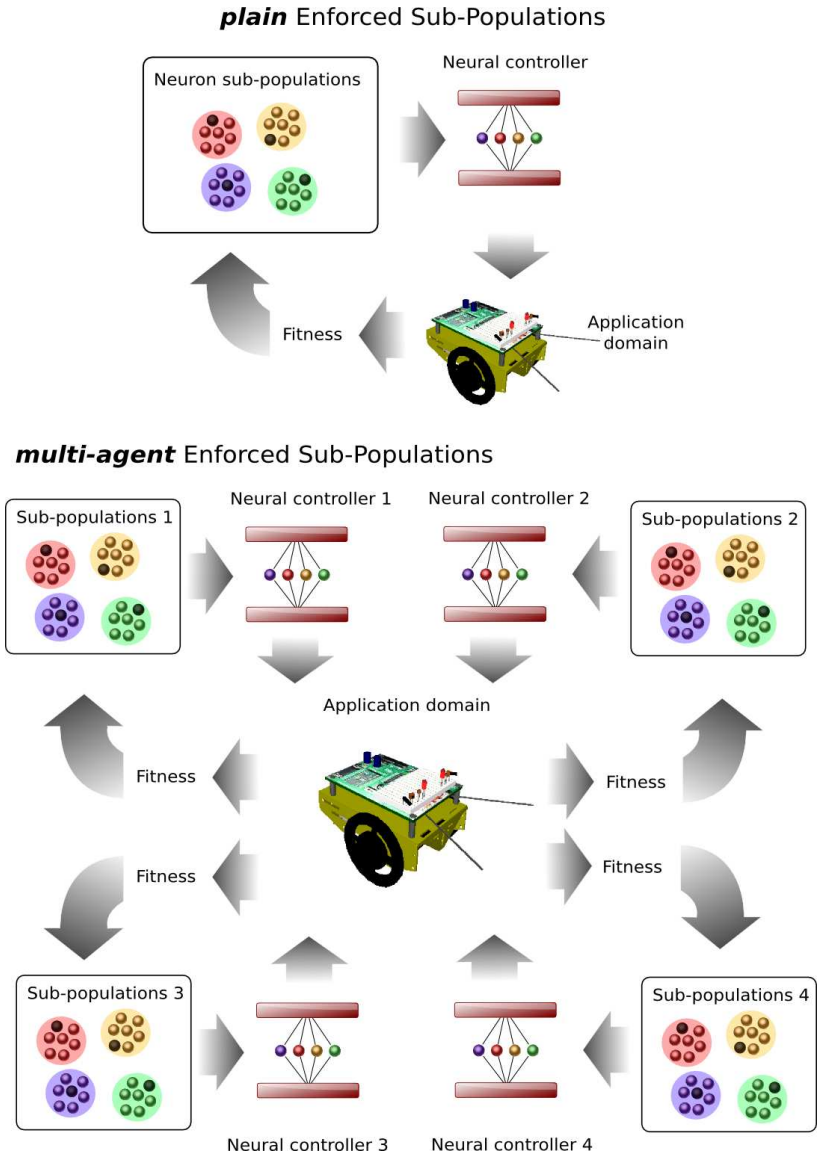


Figure 4.5: Differences of a controller for the ESP and for the multi-agent ESP when controlling a simple robot agent composed of two sensors and two motors.

Figure 4.6: Encoding scheme used in a DAIR controller composed of n networks with k connections each. Each weight for each the neural net is directly included in the genotype. Weight W_{ij} encodes the j-th weight of the i-th network.

Table 4.1: Table with the recommended use of the different modularity types, based on the complexity of the robot and the complexity of the task.

Complexity	Low cmplx. task	High cmplx. task
Low cmplx. robot	Monolithic OR Tactical	Strategic OR Tactical
High cmplx. robot	Tactical	Strategic AND Tactical

approach has been selected. Hence, all the IHU's participating in an evaluation will share the same fitness.

Encoding scheme

A direct encoding was selected as the encoding scheme for this architecture. Accordingly, each of the ANN weights will be directly encoded in the genotype as a genetic trait (see figure 4.6). The decision to use this simple encoding is in order to maintain a low complexity level of the architecture implementation. The job of designing DAIR architecture with more efficient encoding schemes is left for the future.

Genetic operators

The ESP algorithm uses the two most common genetic operators; namely one point cross-over and mutation. Mutation rates are kept low.

4.2.5 Combining tactical and strategic modularity

Tactical modularity may be used in a standalone manner or combined with strategic modularity. In fact, the architecture was planned for its use in complex controllers where both types of modularities should be used to generate a really complex robot behaviour. The use of one type of modularity does not prevent, in principle, the simultaneous use of the other type of modularity. A summary of recommended modularity use is provided in table 4.1.

When the robot to use is simple both strategic or tactical modularization can in principle be used. In such cases, selection of modularity type should be based on the complexity of the problem. When the problem is simple and the number of devices is low, even a monolithic controller can be advised. For complex problems, strategic modularity alone may be the best option. Selecting

a monolithic or strategic in lieu of a tactical modularity should depend entirely on what level of proficiency can be expected from the controller; when only a good enough controller is required, then monolithic or strategic may do the job. However, as will be seen below, tactical approaches usually obtain better performance than monolithic and strategic approaches. The price to be paid is a longer evolutionary time, this due to the higher number of weights to evolve.

In the event that the robot is complex (i.e., with a large number of devices), and the task to solve is simple, a tactically modular controller alone may be the best option. When the task at hand is very complex and the number of elements is large, then a combination of strategic and tactical modularization may be required.

When combining both approaches in one neural controller a strategic modularization should first be performed in order to identify the different sub-goals required for the implementation. This step can be processed using any of the architectures or approaches already existing in literature for this purpose. Once the strategic modules have been decided and their functionality specified, tactical modularity should be applied to divide each of those strategic modules into tactical modules. The number of tactical modules for each strategic module will depend on the elements (i.e., robot devices) that participate in the resolution of the specific sub-goal. Figure 4.7 shows an example of a highly modularized controller that uses both types of modularity.

Next, each strategic module is trained, that is, tactical modules composing a strategic module. Two training approaches are available:

1. Use a common fitness function and train all of the strategic modules at once.

2. Use a different fitness function per each strategic module, thus evolving each module separately.

4.2.6 A simple example of application

This section demonstrates the application of the DAIR approach to a simple robot composed of two sensors and two actuators. A tactically modular controller will be created to perform a *contour following* behaviour around an object. The same behaviour will also be evolved for a monolithic controller, in order to compare performance (deeper comparisons of the DAIR architecture with other architectures will be shown in the next chapter). Both processes will be performed in the Webots simulator.

The setup

The Webots simulator was used to design the experimental setup. Simulation basically consisted of a open space (see figure 4.8), with a large object in the centre. The robot will be randomly placed inside the open space with a random orientation, but close enough to the object to be able to reach it in a couple of

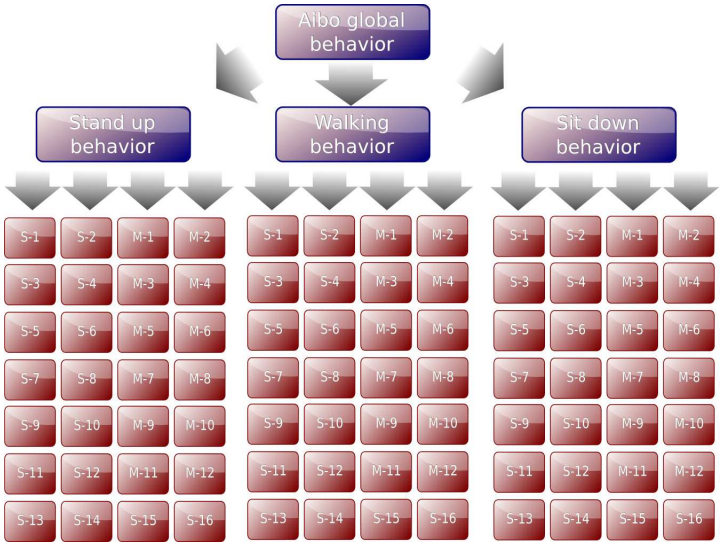


Figure 4.7: A highly modularized robot controller which implements both types of modularization in a single neural controller; for the Aibo robot. The robot should stand up, walk to the other corner and sit down again. Each blue box indicates a strategic modularization. Red boxes are tactical neural modules, - each one containing a neural net. Each strategic module is composed of 28 tactical modules (namely, IHU modules implementing neural nets).

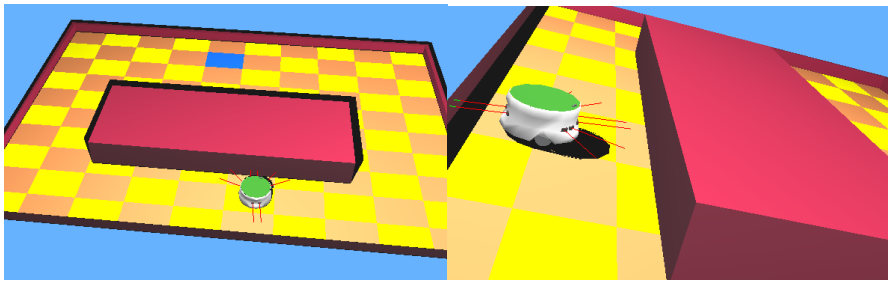


Figure 4.8: The Khepera II robot simulation (left) and its environment (right), used for the example in section 4.2.6. Only two out of the eight infrared sensors available in the robot are used for this example.

Table 4.2: Table of discretization values for sensors

Discretization name	Sensor range
FAR	S > 1000
MEDIUM	500 > S > 1000
CLOSE	500 > S > 250
VERY_CLOSE	250 > S

Table 4.3: Table of discretization values for speed

Discretization name	Speed value
FULL_FORWARD	1.0
HALF_FORWARD	0.5
STOPPED	0.0
HALF_BACKWARD	-0.5

running steps. Its task will be to look for the object and move around it following its contour. To keep the experiment simple only two IR sensors will be used, and the control system will not include integration of information over time. This means that the robot will know nothing about its past and will decide what to do next based only on its current sensors' state (Markov Decision Process condition).

The robot

The robot used for the experiment is a software simulation of the Khepera II robot (figure 4.8). This robot is composed of eight infrared sensors and two wheels (amongst other features). For this simple example only two of the infrarred sensors were used; one of the sensors that points to the front, S_Y, and the left sensor situated on the extreme left side, S_X. The first sensor measures the distance between the robot and the object in front of it, while the second sensor measures the distance between the robot and the object situated to its left.

Simulating real life conditions, sensors have been modeled to be able to detect objects at a range of between 3cm and 20cm. Anything out of this range will not be detected, so it is possible for the robot to be in front of an object and not detect it because it is either too near or too far. So as to keep the whole system simpler the detection values of the sensors have been quantized, allowing only four possible values: FAR, MEDIUM, CLOSE and VERY_CLOSE (see table 4.2). The range of velocities has also been quantized for the motors: FULL_FORWARD, HALF_FORWARD, STOPPED and HALF_BACKWARD (see table 4.3). Both motors and sensors are modeled by a noisy schema defined in the simulator.

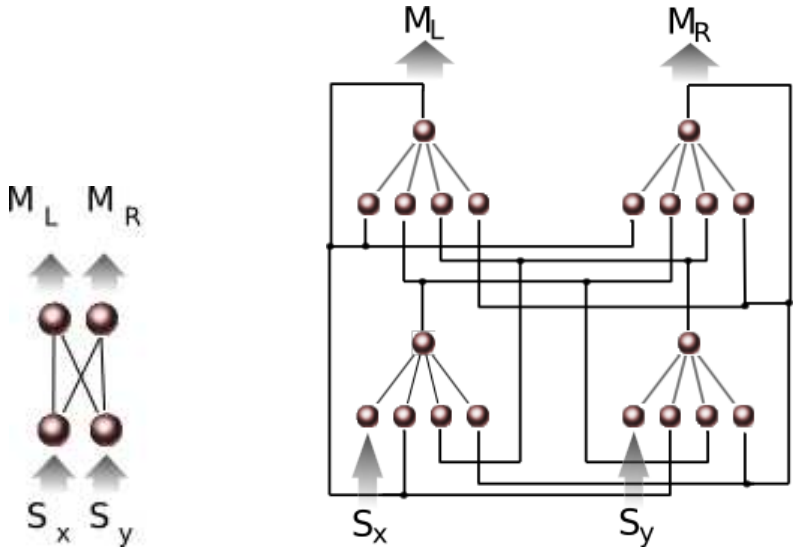


Figure 4.9: The neural architectures used for both controllers tested in the contour following experiment. The figure on the left displays the neural controller used for the *monolithic controller*. The figure on the right shows the DAIR controller.

The evolutionary process

The integration of the evolutionary process with the simulator works as follows: the evolutionary algorithm functions as a supervisor controller of the simulation. After initialization of all populations, a genotype is created for evaluation from the pool of genotypes. The genotype is expressed (converted) into the phenotype (that is, the neural network or group of neural networks). This phenotype is then sent to the robot controller, which will use it in the simulation for an amount of time (arbitrarily decided by the experimenter). Once the time has elapsed a fitness value is generated by the controller based on its performance; this value is then sent to the supervisor, indicating the fitness obtained by the genotype that was tested. A new genotype is then expressed and sent to the robot controller, and the entire process repeated once again. Technical details of how this procedure is actually implemented in the Webots simulator can be found in Appendix B.

The same process is performed for both the single network controller and the DAIR controller, the only difference being the expression of the genotype into the phenotype; one single network in the monolithic case, and four networks in the DAIR case. The following fitness function was defined to obtain the *find*

Table 4.4: Parameters used for the ESP algorithm in the evolution of the contour-following controllers.

Parameter Name	Parameter Value
Subpopulations	1 (monolithic), 4 (tactical)
Size of subpopulations	40
Mutation rate	0.4
Number of trials	10
Stagnation	10
Number of evaluation time-steps	200
Time step (ms)	100
Number of generations	200

and orbiting behaviour:

$$fitness = \begin{cases} +stored \quad when \quad \begin{array}{ll} & S_X = \text{CLOSE} \\ \wedge & S_Y = \text{FAR} \\ \wedge & V_{left} \neq \text{STOPPED} \\ \wedge & V_{right} \neq \text{STOPPED} \end{array} \\ -1 \qquad\qquad otherwise \end{cases} \tag{4.1}$$

where *stored* is a variable with initial value = 1. At any time step that the robot is following a contour *stored* value is incremented by one unit. Whenever the robot loses contact with the object *stored* value drops to 1. S_X is the discretized value for sensor X, S_Y is the discretized value for sensor Y, V_{left} is the discretized velocity of the left wheel, and V_{right} is the discretized velocity of right wheel. This fitness function is calculated at each time step of the simulation. It will increase the reward in one unit at each time step that the robot is running forward with both wheels, detecting an object on its left at CLOSE distance, and detecting nothing in front of it at that time step.

Some values required for the algorithm are presented in the 4.4 table: the number of subpopulations is equal to the number of output neurons; the size of sub-population parameter indicates the number of neurons available for its use in each subpopulation; the mutation rate expresses the rate at which neurons are mutated; the number of trials indicates the mean number of times that every neuron of every subpopulation must be tested before recombination is started; and the stagnation parameter defines the number of trials without improvement before delta-coding is invoked. The number of steps indicates the amount of step times that an evaluation is run until it reaches the maximum fitness. The number of generations shows the number of times that the evolutionary algorithm is applied. The same experiment was run 7 times for 150 generations each run. The final result is shown in figure 4.10 as an average of all the 7 runs. Additionally, the fitness evolution of each of the 7 runs can be viewed in Appendix C.

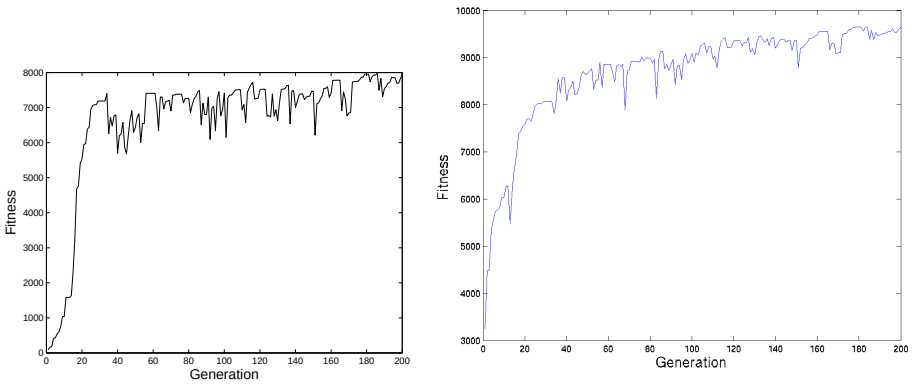


Figure 4.10: Evolution of the average fitness through generations for the monolithic (left) and the DAIR (right) controllers.

Results

After the evolutionary process is finished, it is observed that the behaviour obtained in both controllers is very similar. Basically, the main behaviour is composed of two sub-behaviours; one behaviour searching for the central object, and another behaviour for moving around the object. The first behaviour appears when the robot senses nothing, that is, all the IR sensors of the robot sense no obstacle. Thus, the behaviour obtained for this situation is a circular movement which assures the robot that at any point, it will find an obstacle, independently of its initial orientation. Once the robot detects the object with the S_Y sensor it starts turning until S_X detects the object and S_Y does not. The robot then starts a forward movement, which moves the robot along the contour of the object in counter clock-wise rotating direction. This behaviour allows the robot to regain contact with the object once it loses its detection on reaching the corner.

A single neural net was used (figure 4.9-left) for the centrallized controller. The required behaviour was obtained after the 200 generations, with the average maximum fitness obtained being 7996. For the distributed controller, instead, four different neural networks like the one shown in figure 4.9-right were used; one for each IHU. All the neural networks were evolved at the same time with the same parameters as those in the 4.4 table, as well as the same fitness function. In this case, the contour-following behaviour was evolved after the 200 generations with an average fitness value of 9689.

When comparing results for the monolithic and the distributed experiments it is observed that the behaviour obtained shows no observable visual differences. A few differences can be observed from a technical point of view however; firstly, the averaged number of generations required to obtain the same behaviour was smaller in the case of the distributed controller, and secondly, the maximal fitness obtained by the distributed architecture is greater than that obtained by

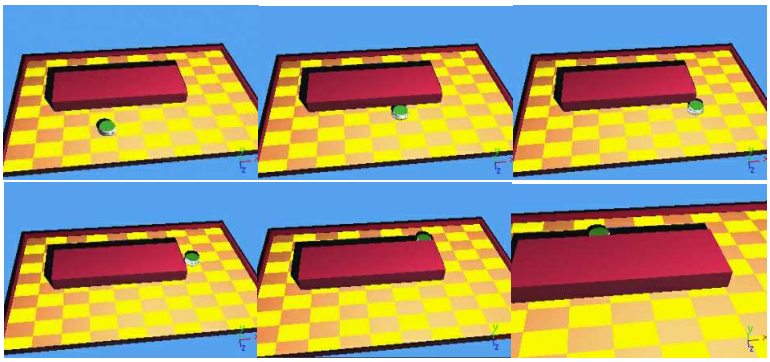


Figure 4.11: Sequence of images showing the contour-following behaviour obtained.

the monolithic approach. The distributed approach learnt faster (in terms of generations) and better (in terms of fitness value) even though the number of weights to evolve in the distributed case was four times greater than the number of weights for the monolithic controller. Due to the higher number of weights to evolve, a distributed generation lasted more time (in terms of number of seconds) than a monolithic generation.

The sensor sub-agents' real job

When using four IHU's to control the robot the question about the necessity of using IHU in the sensors arises. Since its only job is to receive the value from the sensor and deliver it to the rest of neural networks, the question is whether it is really necessary to use these elements in such passive elements. Is the IHU attached to the sensor's performing any job? From an intuitive point of view, it seems clear that attaching a processing element to the sensor output should add value to the behaviour of the robot, though it may result in the cost associated with its evolution reducing the overall performance of the whole robot controller. Hence, to answer these questions an additional experiment was performed in the contour-following behaviour setup.

In this experiment, a distributed controller with only two IHU's, - one per each motor was created, that is, no IHU were used for the sensors (see figure 4.12). Output from S_X and S_Y sensors was directly connected to the neural networks inputs of the actuators IHU. The same evolutionary process was performed for the evolution of the contour-following behaviour. Results showed that the resulting robotic agent was able to evolve the required contour-following behaviour. Its fitness dropped a bit from the maximum averaged value obtained with DAIR (an average of 8019 for the controller without sensor IHUs, against 9689 for the four agents distributed controller), but it was over the maximum averaged value obtained by the monolithic controller (whose maximum averaged value was 7996). From this result, it can be observed two things: first, a *medium*

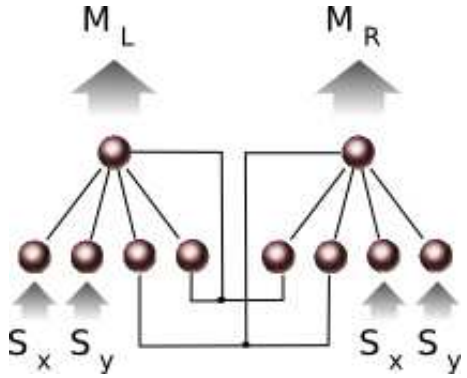


Figure 4.12: Neural controller for the Khepera II robot, where the sensor IHU's have been eliminated, and a direct connection between the sensors and the actuator IHU's designed.

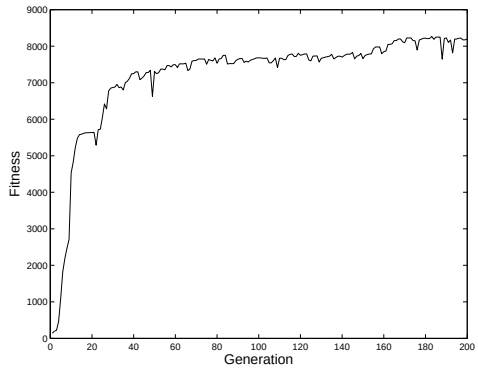


Figure 4.13: Fitness function of the experiment without sensor IHU's.

modularized controller (that is, the DAIR withour sensor IHUs) results in a fitness improvement when compared to the non modularized controller. Second, there is a drop in fitness value from having a fully modularized controller to the half modularized one. We can infere then that sensor IHUs are really performing a useful function, that is, they are learning something that improves the agent behaviour. This first illustrative result will be corroborated in the next chapter in a more complex setup (see section 5.3).

Combining strategic and tactical modularity

In order to demonstrate how strategic and tactical modularity can be combined to generate a single controller an application example of the generation of the contour-following behaviour is provided. If a strategic approach is used to identify the required behaviours for the contour-following behaviour, it has been

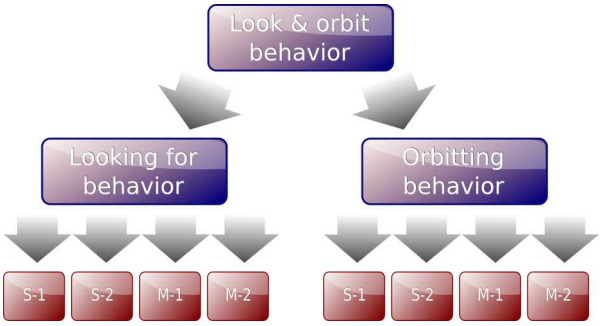


Figure 4.14: Combined strategic and tactical controller for the Khepera II robot.

observed that two simple behaviours are required: one *searching for* behaviour which will look for the central obstacle, and one *orbitting* behaviour that will move around the object. There are therefore two strategic modules to implement. Each of those modules should be *modularized* using the tactical approach afterwards. This means that each strategic module will be composed of four neural networks to be evolved. The schematics of the final controller is shown in figure 4.14. The controller works as follows: initially, the *searching for* controller takes control of the robot. This strategic module controls the robot until the obstacle is detected, either with sensor S_X or S_Y. The *orbitting* strategic module then takes control. At this point, the *orbitting* module will turn the object clock-wise until the S_X sensor starts to detect the object. Switching between modules is manually programmed into the controller program, and is based on the status of the IR sensors; when S_X and S_Y are not activated by the presence of any obstacle, the *searching for* module takes control, and when any of the sensors is activated, the *orbitting* module takes control.

Given the described setup, the evolution of the controller is performed. Parameters for the evolution are shown in table 4.4. Due to the strategic division made, it is possible to train each strategic module in separated evolutionary processes. This procedure would require the generation of a fitness function corresponding with the behaviour required for that strategic module for each module. However, when the robot used and the behaviours required are both simple, it is possible to evolve all the tactical modules of all of the strategic modules in one single evolutionary process. As a drawback, evolving all the strategic modules at once can result in a less efficient controller due to the effect of genetic linkage between modules. This effect is better described in section 5.4.

For this case, and due to the simplicity of the robot and task to be evolved, the two strategic modules are evolved at once: all the tactical modules of both strategic modules evolved in the same evolutionary process and with the same fitness function. As fitness function, equation 4.1 was used. The result was a correct contour-following behaviour, where no differences with the *only tactical* controller were observed. The experiment was only performed once, - simply

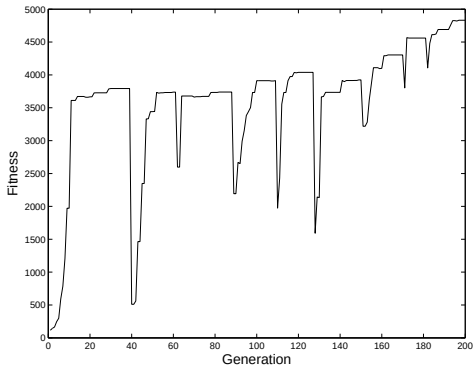


Figure 4.15: Fitness obtained for the combined strategic+tactical contour-following behaviour

to use it as a proof of concept, and no averaged results are thus provided (see figure 4.15 for obtained fitness).

4.2.7 Discussion

Results obtained for the contour-following example illustrate how the approach works for the control of robots. They show that the DAIR controller performs better than the centralized monolithic one when evaluated on the same simple task and (simple) robot. However, there are no conclusive arguments that could justify the use of one over the other, as both controllers were able to present the required behaviours with just small differences in fitness performance and number of generations required. In some cases, when the improvement in the fitness value does not affect the perceived behaviour, the increased degree of complexity introduced by the strategic modularization may well not be worth it. Experiments have shown that controller performance is improved with the use of sensor subagents.

Finally, one can observe the strong similarities that the tactical modularization approach shares with the schema-based control approach in [Arbib, 1992, Arkin, 1998, Murphy, 2000]. In this theory, the control of the robot is based on *perceptual schemas* and *motor schemas*. Perceptual schemas are linked to sensors, and motor schemas are linked to actuators. Basically, a schema consists of the knowledge of how to act and/or perceive, and the computational process by which it is used to accomplish a task. The *perceptual method* of a perceptual schema takes the sensor input and transforms it into a data structure called a *percept*. The motor schema has at least one method (known as the *motor method*) which transforms percepts into an action. So, a tactically modular controller may be seen in terms of perceptual and motor schemas, where each sensor has a *single* perceptual method associated (producing a single percept), and each actuator has a single motor method which takes into account all of

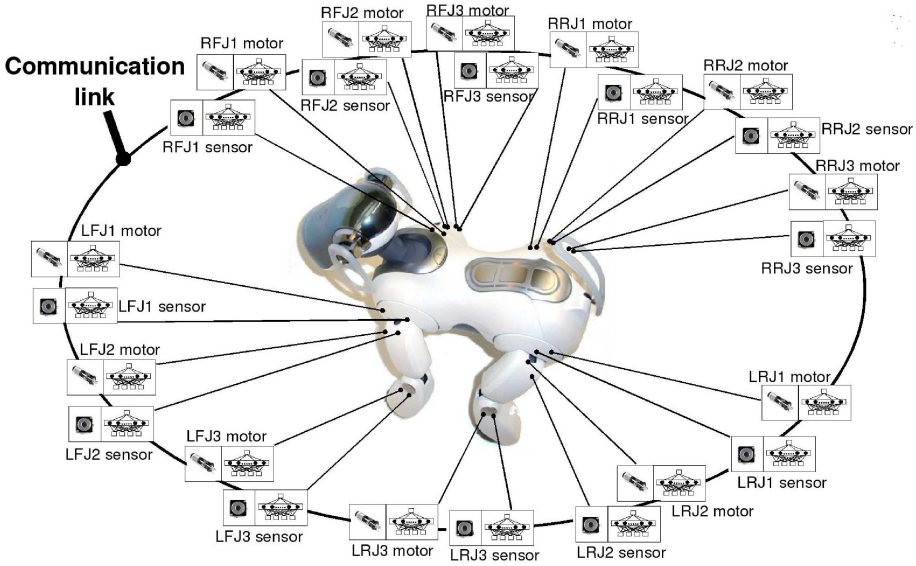


Figure 4.16: A typical application of the tactical modularization in the Aibo robot. A neural module is designed for each sensor and actuator that will take part in the required behaviour.

the percepts generated in the robot.

4.3 Using the architecture in a complex robot

The Aibo robot has been selected for a validation of the control architecture on real robots. Aibo is a complex robot with 18 degrees of freedom and multiple sensors and actuators that requires a good coordination between them to achieve any simple movement task. This robot will be used in further experiments throughout the length of this thesis as representative of a complex robot.

The aim for this stage was to check the proposed control architecture on such a complex robot in three different tests. Each test consisted in making Aibo perform a determined behaviour. Several combinations of simulation and real robot were used in order to test the architecture in all the possible situations. The first task was the generation of a *remaining in a standing position*, evolved only in simulation; the second task was to learn how to *push into the ground* with one leg, performing the whole evolutionary process in the real robot; and the third task was to generate a *standing up* behaviour from a lying position, evolving in the simulator and transferring the result to the real robot.

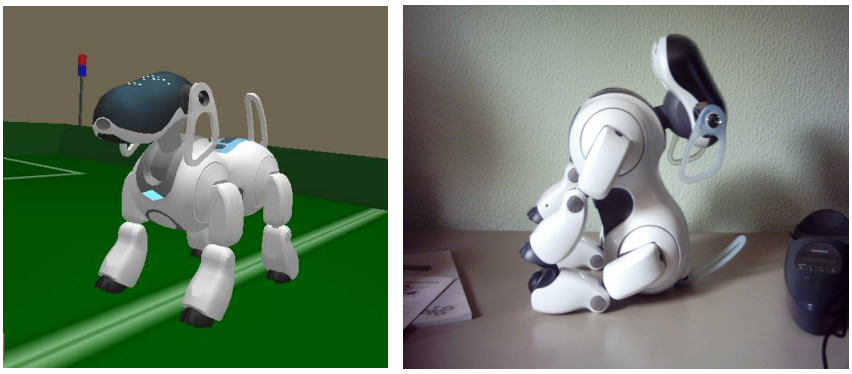


Figure 4.17: Aibo pictures of the ERS-7 model, for the simulation (left) and for the real robot (right).

4.3.1 The robot and its working environment.

Aibo is a quadruped robot with several existing models (ERS-110, ERS-111, ERS-210, ERS-220, ERS-7) [Fujita and Kitano, 1998, Fujita, 2001]. The ERS-7 model, the newest, was used in our experiments. The ERS-7 has 18 degrees of freedom (DOF), and each leg has three DOF (three motor joints per leg), as well as the head. The two additional DOF's correspond to the tail of the robot. It is equipped with a sensor in each of the joints that allows it to sense the current position of the joint. Each leg has three different types of joints; we will refer to them as: J1 for the shoulder joint, J2 for the elbow joint and J3 for the knee joint. There is a paw sensor at the foot of each (Aibo) leg which indicates whether Aibo is touching the ground or not, and it is additionally equipped with three accelerometers, three infrarred sensors (for the detection of obstacles), a colour camera and two microphones.

When the evolutionary process is performed on-line (that is, on board the robot), the evolution of some behaviours for Aibo can last for days. Furthermore, initial behaviours obtained during the early stages of some evolutionary processes may damage the robot, and a simulator of the Aibo robot is used in some cases to avoid this type of situation. Webots software by Cyberbotics [Michel, 2004, Téllez and Angulo, 2007] was selected as simulator. It allows the simulation of most of the robot's features. It also provides a mechanism for direct control of the real robot from the simulator based on a server-client architecture, where the robot acts as the server and accepts connections from the client computers, requesting different sensor data or sending commands to the motors. The simulator also contains a transfer feature of evolved controllers from the simulator to the real robot. Hence, once a controller has been generated and tested on the simulator it can be directly transferred to the real robot[1].

[1]At the time of beginning this research the Aibo simulation of the ERS-7 model was not included in the simulator (only the ERS-210 model was supported at the time). A cooperation

4.3.2 First test: remaining in a standing position

The architecture's first test was to make the robot remain in a standing position. The robot was set up in an initial standing position in the space and allowed to act using its control networks. The goal here was to make it learn how to keep up as high as possible with the least amount of joint movements as possible (dynamical balance). Even though the task is simple the controller still has to learn it since it will not be performed off-the-shelf; the reason is that when a controller is connected to the joints of a robot, ANN's will send commands to the actuators all the time, and the joints will keep on moving unless the associated nets decide not to move the robot. A continuous movement of the joints could lead to strange robot positions and eventually make the robot fall to the ground. In this task then, the robot must learn how to keep a stable high position until the end of the evaluation time.

Setup

The following sensors and actuators were involved in the control of the robot:

- Actuators: four shoulder joints (J1), four elbow joints (J2), four knee joints (J3). These are all motors that move Aibo's legs and determine its position in the allocated space.
- Sensors: four shoulder joint sensors, four elbow joint sensors, four knee joint sensors and four paw sensors. These are the sensors that indicate the state of the joint motors (actuators). The last four paw sensors indicate the state of the feet paws, and are switched on when the feet touch the ground, and off when not touching. Additionlly, three accelerometer sensors (X,Y and Z) were used to determine Aibo's position.

As a total 31 sensors and actuators are implied, this same number of IHU's are thus required for the generation of a tactical-only DAIR controller. In line with the architectural specification, all the nets have the same number of inputs and outputs; 31 inputs and 1 output. For the hidden units we have selected 8 hidden units based on previous experiments. Values obtained from sensors were quantized, thus allowing a precision of one degree, i.e. the raw output from the sensor was quantized to the closest integer angle before providing it to the IHU. This quantization was required in order to prevent undesirable oscillations and never ending training. Network outputs of sensors were quantized in the same way; providing only three possible values:

- increase the joint angle by 0.05 radians
- decrease the joint angle by 0.05 radians
- not move the joint.

was initiated with Cyberbotics in order to create the new model simulation. More details of the work done at Cyberbotics (which was required for the experiments conducted in this thesis) are provided in Appendix A. Final results obtained in terms of the measured similarity between the model and the real robot can be read in [Holh et al., 2006, Téllez and Angulo, 2007].

Table 4.5: Parameters used for the neuro-evolution algorithm in the Aibo tests

Parameter Name	Parameter Value
Subpopulations	8
Size of subpopulations	40
Mutation rate	0.4
Stagnation	20
Number of trials	10
Number of evaluation steps	300
Time step	96 ms

Evolutionary parameters

Parameters used in the evolutionary algorithm were selected based on previous experiences (table 4.5). The following fitness function was used for the evolutionary process:

$$fitness = \frac{final\ height}{number\ of\ movements\ +\ 1}$$

where *final height* indicated the height of the robot at the last evaluation step, and *number of movements* counted the number of times that joints changed their position since the beginning of the evaluation. Fitness rewards controllers which achieve a high height whilst moving the robot's joints as little as possible. Evaluation time for each phenotype was 300 time steps.

Results

The experiment was repeated ten times and the evolution of the averaged fitness is shown in figure 4.18. A good enough behaviour was obtained (after averaged generation 14). In initial experiments the fitness function denominator was not included. In those cases it was observed that the best evolved controller achieved the maintenance of a stable robot position, but movement of the joints never ended. The addition of the denominator makes the tremor practically disappear.

4.3.3 Second test: learning to touch the ground with one leg

This test performs on-line evolution of the robot leg (see section 2.2.1). The experiment consisted of controlling one of the real robot's legs in order to make it learn how to push against the ground.

Setup

To perform experimentation the robot was sat down in its charging station, with all its legs set to an initial non-interfering position. Then the leg being

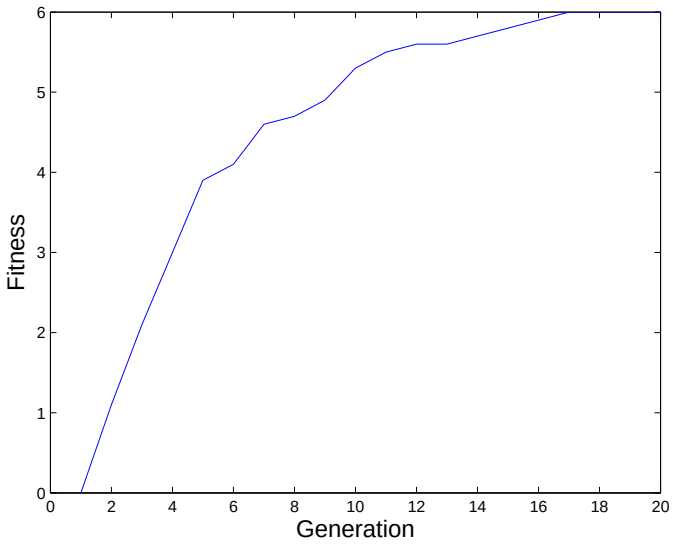


Figure 4.18: Evolution of the fitness for the Aibo remain standing test, averaged after ten evolutionary runs.

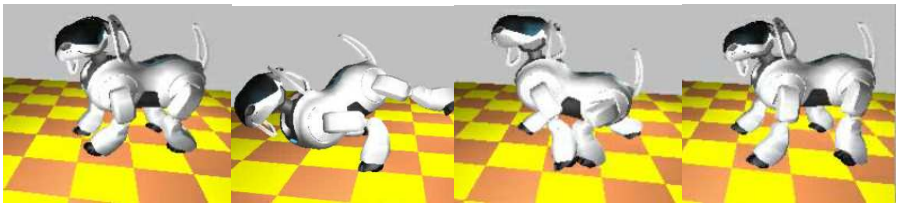


Figure 4.19: Some screenshots for the first test with the Aibo simulator. The evolutionary process starts with the Aibo robot in the situation depicted in figure (1). Figures (2) and (3) indicate two possible situations where the controller tested was incorrect. Figure (4) shows the final position obtained by a successful controller.

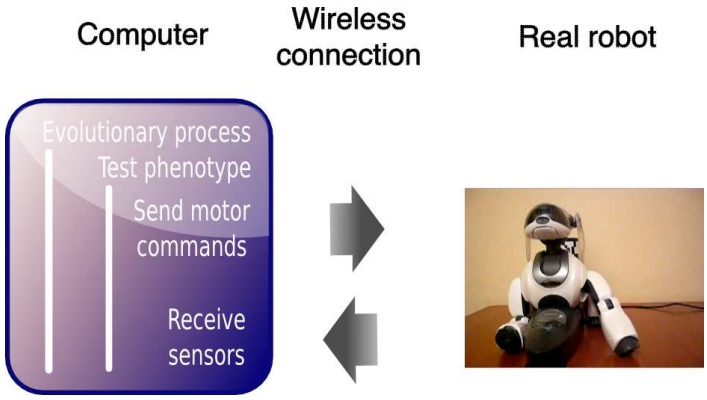


Figure 4.20: Schematics of the evolutionary process in the real robot.

tested (the left fore leg) was moved to a random initial position and control was transferred to the group of neural nets. The desired final position of the leg after 300 time steps should be any position as long as the paw sensor of the leg is activated (which was only possible when pushing against the ground), and the robot does not falls down (due to pushing too much).

In this case the evolutionary program was executed on a PC, implementing the same evolutionary algorithm as in previous section. When a controller is generated for evaluation it is also executed on the same computer, but the controller is fed with real data coming from the robot sensors, and sends motor answers to it as follows: the computer is connected to the robot by way of a wireless connection. Every 100 ms sensors values are requested by the computer from the robot. These values are then used in the sensor IHU's (which are executed in the PC), and the answers generated by the actuator IHU's (motor commands) are then sent back to the real robot as orders for the motors. A schematics of the process is shown in figure 4.20.

Practical implementation of the experiment included several real world issues; like monitoring the temperature of the robot due to its continuous movement, disabling the battery-status check to allow a continuous flow of energy, or manually returning the robot back to the initial position when it fell down due to strange leg movements. A special mechanism was incorporated that saved the evolutionary state each time that a fault condition was detected (overheating, no battery power, falling down). This mechanism allowed the algorithm to restart at the same point once the fault condition was detected and manually solved.

An IHU was created for each of the motors and sensors that took part in the control of the leg; they were 3 IHU's for the leg motors, 3 IHU's for the motor's leg sensors, one IHU for the paw sensor, and finally, in order to control the inclination of the robot 3 additional IHU's were created, - one for each acceleration sensor (X-axis, Y-axis and Z-axis). Inclination control was necessary in order that Aibo taught itself not to push too hard into the ground,

Table 4.6: Parameters used for the second test in the real robot.

Parameter Name	Parameter Value
Subpopulations	5
Size of subpopulations	40
Mutation rate	0.4
Number of trials	10
Stagnation	20
Number of steps	300
Time step	96 ms

thus preventing it from falling to one side. That makes a total of 10 IHU's; 3 of them controlling actuators and 7 controlling sensors. For each IHU a feedforward neural network was used, composed of 10 inputs, 1 output and 5 hidden units.

The output of the actuator IHU's was discretized to allow only three possible answers: increase the current joint angle by one degree, decrease the angle by one degree, or do not change current motor position. Thus, in every evolutionary generation, the training program generated a set of 10 IHU's that made up the controller of Aibo's leg. The process was repeated for 300 time steps.

Evolutionary parameters

The evolutionary parameters used for the experiment are summarized in the 4.6 table .The fitness function used was very simple:

$$fitness = \begin{cases} 1 & \begin{array}{c} when\ paw\ is\ activated \\ \wedge\ -60 < accel_x < 60 \\ \wedge\ -290 < accel_y < -100 \\ \wedge\ accel_z < -900 \end{array} \\ 0 & otherwise \end{cases}$$

The fitness function rewards those controllers where the paw sensor is activated at the end of the evaluation time, and the robot does not fall down. The term *when paw is activated* measures the status of the paw at the end of the evaluation time. The $accel_k$ terms in the fitness function indicate the values of the accelerometer sensors in the three components (for k=x, y and z). Their limit values included in the fitness function were obtained by measuring them under controlled conditions, and allowed to determine when the robot felt down.

Results

Due to both the simplicity of the fitness function used and the large range of movements that the leg can perform when the evolutionary process begins it was not possible to evolve a correct controller for the task. During the first generations of the evolutionary process, most of the controllers received a zero

Figure 4.21: Sequence of movements with one successful controller. Initial position is selected at random by a prepared algorithm. Final position is decided by the controller, i.e., is not a fixed position.

fitness reward, hence, the evolutionary search cannot gradually proceed towards a controller that finally achieves the desired behaviour. This means that the evolutionary process suffered from bootstrapping (see section 2.2).

A way to solve this problem is to use incremental evolution (see section 2.4.1). Using this method, the robot is taught to do the task by teaching it a set of easier versions of the final task. This method is used to reduce the searching space and to allow convergence to a suitable controller in a shorter amount of time. Hence, an additional experiment was performed where incremental evolution was used. The whole evolutionary process was split into seven different evolutionary stages:

1. First, teaching the dog how to touch the ground from a *fixed* position very close to the ground.
2. Once it has learnt how to do this, the initial position was changed to a further (but still fixed) position.
3. Then, the initial position was moved to one even further, but a small random element was also introduced into the initial position.
4. The next evolution involved a position farther from the ground with a bigger random element in the initial position.
5. The addition of randomness was repeated two more times.
6. The final evolutionary step included a completely random position at the start.

By using this method, the robot was able to evolve the ground touching behaviour along 30 generations in 4.2 days. This test was used only as a proof of the concept. Due to the long time required for the test the results obtained were not averaged over several evolutionary runs.

4.3.4 Third test: learning to stand up

The third test performed made use of both simulator and real robot. The goal for this test was the evolution of the DAIR controller in the simulator, and then to transfer it to the real robot in order to validate the result obtained. For this

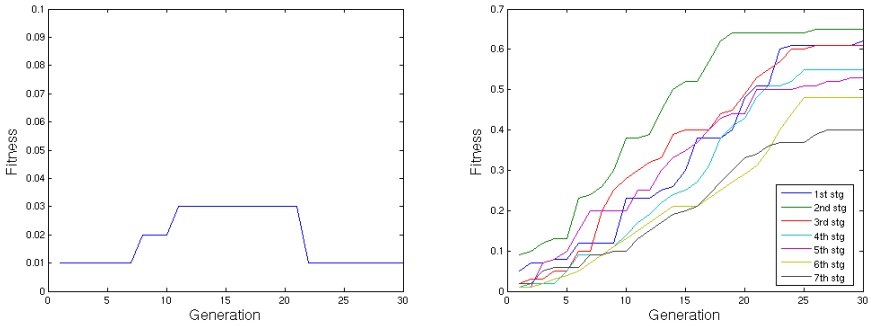


Figure 4.22: Fitness evolution for the second Aibo test. The figure on the left shows the fitness evolution in one single round. The figure on the right shows the fitness for the incremental evolution setup.

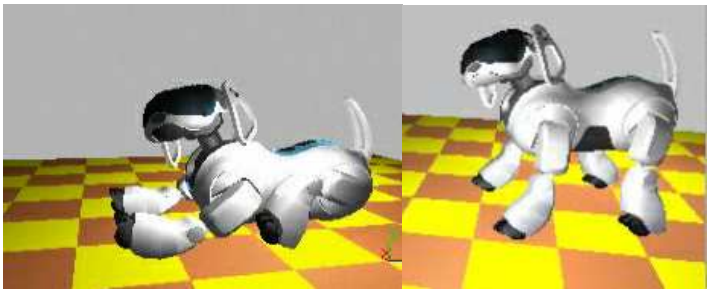


Figure 4.23: The third test specification: the robot starts at the position on the left, and it must change its position to the position on the right.

test, a complete transfer system was created for both the simulator and the real robot. This transfer system contained all the required code for the simulator that allowed a feasible modelization of the robot, and a code for the transference of the controllers from the simulator to the real robot, without requiring any change in the evolved controller. A complete description of the work done in the simulator and server system is provided in Appendix A and additionally in [Holh et al., 2006].

The task selected for the robot was to learn how to change from a *laying down* position to a *standing up* one. This task was especially interesting as it is not straightforward to obtain the sequence of movements which lead to the standing up position of the robot. The reason for this is that a direct movement of the joints from the initial position to the desired position of standing up will end up with the robot falling down. The fact is that when torques to the motors are not performed in a correct sequence the robot falls down.

Setup

The same architecture as in the standing test was used for this test, with 31 IHU's used for the same devices (12 joint motors, 12 motor sensors, 3 accelerometer sensors and 4 paw sensors).

Evolutionary parameters

The same parameters as in the test of the standing test were used for the evolutionary algorithm (shown in table 4.5). The fitness function was defined on a first instance as follows:

$$fitness = final\ height$$

where *final height* is the height achieved by the robot when the evaluation has ended. However, this function in some cases led to situations like the one shown in figure 4.24, where a strange position evolved even though the robot achieved the standing up. In order to eliminate such solutions the fitness function was modified as follows:

$$fitness = paws * final\ height$$

where *paws* indicates the number of paw sensors that are activated at the end of the evolution time.

Results

Figure 4.25-top shows a sequence of movements obtained for the final behaviour. Almost the same behaviour was obtained in the ten evolutionary runs, with a mean number of 10 generations to obtain such behaviour. The maximal mean fitness value obtained was 462 (see figure 4.26). Due to the fact that the height sensor is situated in the head of the simulated robot the final position achieved by the robot is a bit different from the one specified in figure 4.23, since the

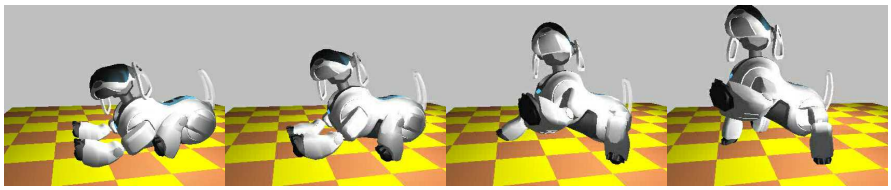


Figure 4.24: A sequence of movements for a strange standing up behaviour evolved with a fitness function without the *paw*s term.

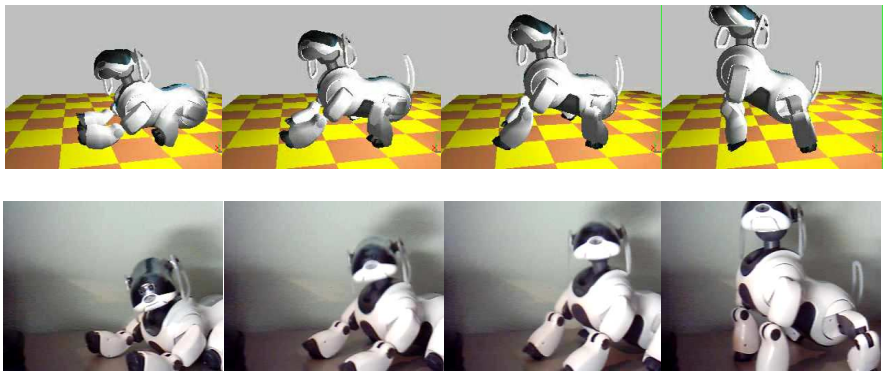


Figure 4.25: Sequence of figures showing the change from a laying down position to a standing up position in both simulation (top) and real robot (bottom).

robot tries to keep its head as high as possible. The final position of the robot was however close enough to the desired one.

Once the behaviour of figure 4.25-top was obtained, the best DAIR tactical controller was transferred to the real robot using the simulator transference facility that we developed. The behaviour observed in the real robot is shown in figure 4.25-bottom, and does not present a significant difference when compared with the behaviour obtained in the simulator.

4.4 Conclusions

In this chapter, two different approaches to modular neural control have been identified; one which performs modularization at the behavioural level (strategic), and another one which performs modularization at the device level (tactical). We identify strategic modularization with already existing architectures for the creation of modular controllers. We define the concept of tactical modularity as a new one, which deals with modularization at the level of the elements that actually perform a sensorimotor coordination. Based on this idea, a distributed control approach for the control of complex robots has been created. The approach includes sensorimotor coordination as the central tenet, massive

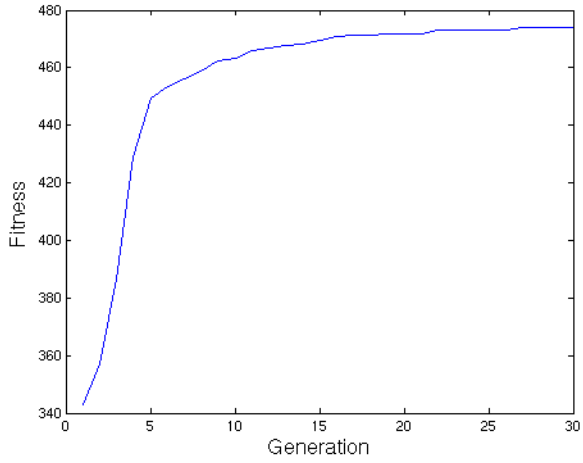


Figure 4.26: Evolution of the fitness for the third Aibo test, averaged after ten evolutionary runs.

modularity, double closure and bias introduction.

Strategic and tactical modularity are not excluding one from the other. For a given controller the use of one type of modularity, the other type or a combination of both may be more convenient. Selection of one type or another should depend on the complexity of the robot and the task to perform.

The use of tactical modularity has proved to work on simple and complex robots, in both simulator and in real robot, and with different combinations of evolutionary processes (only simulator, only real robot, or a combination of simulation and real robot). Experiments have shown that the defined architecture works in those situations. Some preliminary results show that the use of tactical modularity when compared with a monolithic controller may imply an improvement in the maximal fitness reached. An in depth study of this question is provided in the next chapter, where the tactical approach is compared with several other architectures and a benchmarking is provided.

It is like comparing champagne with cognac. No-with Coca-Cola
Maria Callas

5 Architecture benchmarking

The novel methodology presented in the previous chapter will be extensively tested and compared with current approaches in this chapter. Khepera robot simulations will be used as test bed. The experiments will consist of implementing a DAIR control system for the Khepera robot while it performs a test bed task. The DAIR architecture's performance will be compared with the results obtained by state of the art neural architectures on the same task.

The selected test bed task is referred to as the *garbage collector* experiment, and it follows the description given in [Nolfi, 1997]. In this task, a Khepera robot is placed inside an arena surrounded by walls (figure 5.1); the robot looks for sticks that have been randomly distributed in the space, grasps them, and takes them out of the arena. This test bed is particularly interesting as it is not a simple direct task like avoiding enemies or following walls, but requires a more complex behaviour; evolution of classification from scratch; a complete change in the robot's behaviour based on a single sensor measure. When the robot is not carrying a stick in the gripper its behaviour involves avoiding walls, looking for sticks, approaching them, and finally picking them up. When the robot is carrying a stick the opposite behaviour is needed, i.e. it should avoid the other sticks and approach walls in order to release the carried stick outside the arena. The concepts of strategic and tactical modularity in all their combinations (monolithic controller, strategic only, tactical only, strategic and tactical) will be tested.

5.1 The garbage collector problem

All the experiments described using the Khepera robot were carried out under simulation. As a simulator, the commercially available Webots simulator by Cyberbotics was selected. As well as having the simulation of the Aibo robot used in the previous chapter, Webots contains the simulation for many other robots, including the Khepera robot.

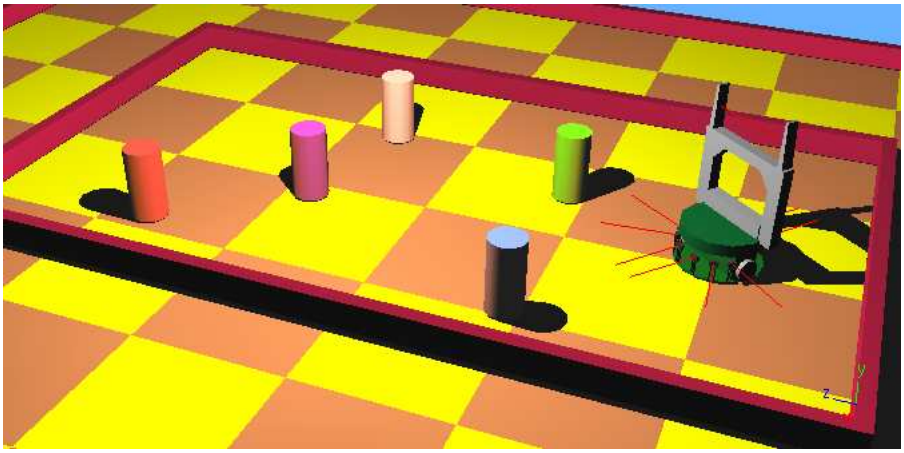


Figure 5.1: Simulation of the garbage collector problem in the Webots simulator.

5.1.1 The environment

The complete set-up for the garbage collector experiment is shown in figure 5.1. It consists of a rectangular, 60x35 cm arena which is surrounded by walls. Inside the arena there are five cylindrical sticks which will perform the role of the garbage; each stick has a diameter of 2.3 cm and is randomly positioned inside the arena for each new epoch. To simplify the problem the sticks are placed with enough separation between each other and the wall to avoid overlapping detection. This means that robot sensors cannot detect several sticks, or a stick and a wall at the same time. Thus, if sensors are activated at any time, it would mean that the robot is detecting either a wall or a single stick. Finally, at the beginning of each controller evaluation, - otherwise known as an epoch, the robot is randomly placed inside the arena.

5.1.2 The robot

The Khepera robot is a miniature mobile robot developed at EPFL in Switzerland. It has a circular shape with a diameter of 55 mm, a height of 30 mm, and a weight of 70 g. It is supported by two motorized wheels, with two small Teflon balls which serve as balancing points.Wheels are controlled by DC motors with an incremental encoder, and they can move in both directions, backwards and forwards. In addition, the robot is equipped with a gripper with two degrees of freedom. The simulation of the Khepera includes the simulation of the Khepera gripper that the robot uses to grasp objects (see figure 5.2). The Khepera gripper is composed of an arm that can be moved at any angle from vertical through to horizontal, and two gripper fingers that can assume either an open or a closed position. The gripper is also incorporates a sensor that indicates the presence of an object between the fingers.

The robot contains eight infrared (IR) sensors; six at the front of the robot,

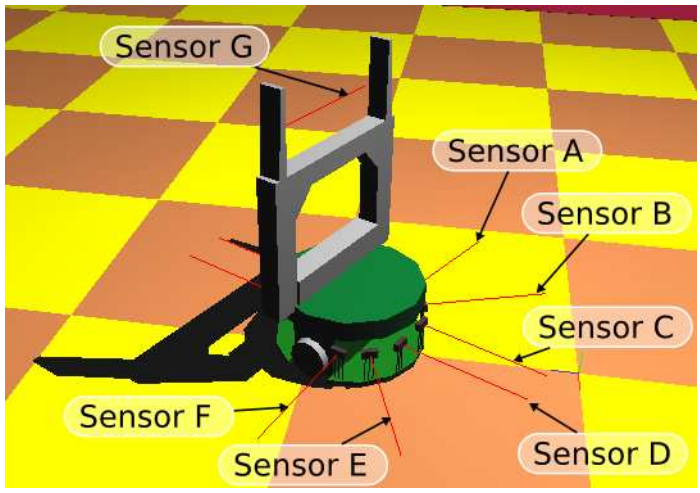


Figure 5.2: Simulation of the Khepera robot including the Khepera gripper. Red lines indicate the range of the infrared sensors.

and two at the back. One presence detector sensor is situtated in the gripper fingers. Only the six front sensors and the gripper sensor were used for the garbage collector task. Those have been marked in the 5.2 figure from A to F for the front IR sensors, and G indicating the gripper sensor. IR sensors have a limited detection range (from 0 to 30 cm), and they provide 1024 different values from 0 to 1023 which indicate the presence of an object at the corresponding distance. Objects placed at a distance exceeding 30 cm are not detected at all. The IR sensor model includes a noise model that affects the measurements.

As actuators, the robot has two motors (left and right), and it is also able to control the position of the gripper arm and the status of the gripper fingers (open or closed). In order to keep things as simple as possible the same method as in [Nolfi, 1997] has been used for the control of the gripper turret through the use of two separate procedures which activate a complete behaviour. When either of the gripper procedures is activated the gripper will perform a complete series of movements which cannot be interrupted until they have been finished. The gripper procedures are as follows:

- *pick-up procedure*. This is activated to make the robot pick up an object. As a pre-condition, the robot must have its arm up and the gripper fingers open. When activated, the following movements are performed: move the arm down, and once completely down close the gripper fingers. In case the robot has not captured anything between its fingers, it will open it's fingers again. The procedure ends by moving the arm up. At the end of this procedure, the robot will have it's arms up, and fingers open (in the event that no stick was captured) or closed with a stick in between them (if a stick was captured).

- *release procedure*. This procedure is activated to make the robot release a stick that is between its fingers. The movements are as follows: robot moves arm down, opens gripper fingers, and moves arm up again. It can be activated at any moment, even if the robot is not carrying a stick.

Of course the controller knows nothing, neither about the behaviour of those procedures nor when they have to be activated. The evolutionary process will have to generate the appropriate controller which activates those procedures when necessary. Procedures are activated on an ON-OFF basis, i.e., they will be taken as if they were actuators which can be controlled with two possible values: an *on* value when activation is required, or an *off* value when activation is not required.

Values detected by the IR sensors (a value between 0 and 1023) are linearly encoded as floating point values between 0.0 and 1.0 to become inputs of the neural networks. The activation of both motors by the networks output is transformed into 21 integer values ranging from -10; for a maximum speed backwards, to +10; for a maximum speed forward. The activation of the procedures follows a thresholded output, where a neural network output above 0.5 will mean *procedure activated*, and an output below 0.5 will mean *procedure not activated*. Sampling time in the simulator is 100 ms. The activation of any of the procedures requires 20 time steps where the robot does nothing but the execution of the activated procedure.

5.1.3 The evolutionary process

Experiments used the evolutionary algorithm ESP to evolve the connections of a set of different neural network-based controllers to correctly perform the garbage collector task, that is, to look for a stick while avoiding walls, pick up the stick, and then release it outside the arena whilst avoiding other sticks. The task was considered solved when the robot achieved the release of one single stick outside the arena. It was considered failed if no stick was released at the end of the evaluation time, or if any of the following errors were produced:

- The robot crashed into a wall
- The robot released a stick on top of another stick
- The robot tried to grasp a wall

To compare the performance of the DAIR architecture with other existing ones, the same setup was used for the evolution of 6 different neural architectures, described in the 5.2 section . The ESP algorithm was used to evolve a suitable controller for each of the given architectures. The same evolutionary parameters were used in the algorithm for the evolution of all the architectures; these are described in the 5.1 table .

During the evolutionary process each generated phenotype needs to be tested on the task at hand. Due to the random nature of the process, and since sticks and robot are randomly initialised at the beginning of the test, fitness for a

Table 5.1: ESP parameters used in the evolution of all the architectures tested on the garbage collector problem.

Parameter Name	Parameter Value
Subpopulations	1
Size of subpopulations	40
Mutation rate	0.8
Stagnation	no stagnation allowed

given phenotype is obtained by testing the same phenotype 15 times, that is, 15 epochs. Each epoch lasts for either 200 time steps of 100 ms each, or until a stick is released outside the arena. After each epoch, the fitness for the phenotype at that epoch is stored. The final fitness for a phenotype is calculated as the mean value of the 15 epochs.

A unique fitness function was created for the evolution of each of the architectures (except for the *(e)* architecture , which requires special treatment, as will be described below). The fitness function rewards the controllers that have been able to release one stick outside the arena. Controllers that are only able to pick up one stick but not release it outside the arena are awarded with a lower fitness. Controllers that were able to pick up a stick but released it inside the arena were severely punished. The definition for the fitness function is the following:

$$fitness = \begin{cases} 1 & if\ stick\ released\ outside\ arena \\ 0.1 & if\ stick\ picked\ but\ not\ released \\ -0.5 & if\ stick\ released\ inside\ arena \end{cases} \quad (5.1)$$

Each evolutionary process lasts for 1000 generations and was performed ten times for each architecture. The results presented below show the average fitness value of those ten runs for each architecture. A detailed list of the fitness evolution for each evolutionary run and architecture can be found in appendix D.

Architecture *(e)* is based on a strategic+tactical controller which is composed of two different strategic modules (see section 5.2). Hence, the controller needs to be evolved in two stages with two different fitness functions. In the first stage, the strategic module in charge of avoiding walls and picking up a stick was evolved with fitness function:

$$fitness = \begin{cases} 1 & if\ stick\ picked\ up \\ 0 & otherwise \end{cases}$$

For the second stage, the process evolved the strategic module in charge of avoiding sticks and approaching walls to release the stick, using the following fitness function:

$$fitness = \begin{cases} 1 & if\ stick\ released\ outside\ arena \\ -0.5 & if\ stick\ released\ inside\ arena \\ 0 & otherwise \end{cases}$$

In the same way as experiments in [Nolfi, 1997], a special mechanism was implemented which artificially placed another stick in front of the robot each time it picked one up. This artificial mechanism allowed an increase in the number of situations where the robot encountered an obstacle in front of it whilst carrying a stick. In [Nolfi, 1997] it was observed that evolved controllers were not able to generate a robust behaviour for the avoidance of sticks while carrying another when this procedure was not introduced into the evolutionary process. The reason for this is that because of the small number of sticks in the arena few occasions of encountering a stick while carrying another arise during the evolutionary process.

5.1.4 Additional configuration

When reproducing the garbage collector problem in the simulator, a free interpretation of some minor simulation factors not described in [Nolfi, 1997] was made. These include the following:

- The pick-up procedure is activated to pick a stick up, but no stick is picked up: in this case, an additional term was added to the pick-up procedure; that whenever it detected that its gripper fingers were closed without having a stick in them, it opened it's fingers again, before moving up the gripper.
- ANN's activating both grasping procedures are activated at the same time: preference was given to the pick-up procedure. Other possible solutions would be to randomly select one of the procedures, or select the one with the highest activation value.
- The pick-up procedure is activated while carrying a stick: in this case, we did nothing with the gripper, but the execution time of the activated procedure was discounted from the total number of time steps available for evaluation of such phenotype. This choice results in a *soft* penalization of the controller.
- A stick is picked up and then released inside the arena: this situation was taken into account in the fitness function (see the 5.1 equation).

5.2 Tested neural architectures

Six different architectures were tested with the set-up described above. They represent a set of controller modularity concepts based on current literature as described in chapter 3. The 5.2 table shows a summary of the architectures tested, along with their identifiers used along this text. The first three architectures are those tested in the original experiments in [Nolfi, 1997], which can be compared with the new ones we introduced; they basically implement strategic modularity in one way or another. The last three architectures make use of

Table 5.2: Table of architectures tested.

Architecture identifier	Type of architecture
(a)	Monolithic
(b)	Distal strategic modular
(c)	Emergent modular
(d)	DAIR tactical modular
(e)	DAIR strategic+tactical
(f)	Tactical with no sensor modules

tactical modularity. They are all described below and can be seen ranging from figure 5.3 to figure 5.8.

Architecture *(a)* : monolithic feed-forward architecture with no modularization (figure 5.3). It is a simple 7-4-4 feed-forward network with four units in the hidden layer, seven inputs corresponding to the six infrared sensors and the gripper sensor, and four outputs that control the two-wheel motors and the two gripper procedures. This is a typical single neural network controller used in most evolutionary robotics experiments [Nolfi and Floreano, 2000].

Architecture *(b)* : distal strategic modularity (figure 5.4). This is a strategic modular architecture composed of two strategic modules. Division into strategic modules has been performed using a hierarchical procedure. Since the robot behaviour switches to its opposite behaviour depending on the state of the gripper (carrying or not carrying a stick), two modules were generated: one module for robot behaviour when not carrying a stick and looking for one, and a second module acting when the robot is carrying a stick and tries to release it outside the arena. This division of the controller into two modules has been manually carried out from our own distal point of view. Each module is implemented by a simple feed-forward network without hidden units. The inputs of the modules receive the information coming from the sensors, and outputs encode actions for the motors and procedures. The state of the gripper sensor decides which module is activated at any given time.

Architecture *(c)* : emergent modular architecture, using proximal strategic modularity (figure 5.5). This architecture was defined in [Nolfi, 1997] as *emergent modular architecture*. Sensors are connected to the ANN's inputs and each actuator is controlled by two modules. Each module is composed of a neural network consisting of seven inputs (for the seven sensors) and two outputs; the output of each group thus consists of four units. The two units of the first module produce two possible actuator action commands for each actuator. The two units of the second module determine which of the two units of the first module will actually drive the actuator. Module one generates two possible

commands for the actuator. Module two decides which of the two command should actually be used to drive the actuator. The two outputs of the second module compete, and the highest output value wins and then decides the actual command. This process is performed for each time step. This architecture is interesting in the sense that it is a modular architecture where the combination of modules for the generation of the behaviour is not decided beforehand, but by the same evolutionary process. Thus, the modularity of this architecture is in some sense emergent.

Architecture *(d)* : DAIR architecture using tactical-only modularity (figure 5.6). It is a direct implementation of the tactical modularity concept to the robot and the task, where a unique global behaviour is required to evolve, that is, the garbage collector behaviour. In this case, tactical modularity solves the whole problem by creating one IHU element for each device involved (one tactical module for each device). Since eleven devices are involved, - seven sensors and four actuators, eleven IHU's are required for the construction of the controller. An IHU was created for each of the infrared sensors as well as four IHU's for the left and right motors and the two gripper procedures. Each IHU is implemented by a feed-forward neural net with eleven inputs, no hidden units, and one output (see figure 5.9).

Architecture *(e)* : DAIR architecture using strategic + tactical approach (figure 5.7). This architecture includes the use of both strategic and tactical modularity in a single controller. In this case, the required behaviour is divided into two main strategic modules, each one in charge of one sub-behaviour (as in architecture *(b)*). Each of those strategic modules is then modularized at the device level using tactical modularization, the same as in the *(d)* architecture. This case is a clear example of the application of both the strategic and tactical modular concepts to solve the control problem. Two different strategic modules, each one implementing one sub-behaviour by means of tactical modularity are obtained. The switch between strategic modules is based on the status of the gripper sensor. The training of each strategic module is carried out by separate evolutionary processes, and once evolved, they are combined to design the global controller. For the training of the first module the robot is placed in a random position, and the evaluation ends successfully when the robot picks a stick without performing an error. The second module evaluation starts with the robot in a random place carrying a stick. The evaluation ends successfully when the robot releases the stick outside the arena without performing an error.

Architecture *(f)* : tactical approach without sensor IHU's (figure 5.8). An additional *half-modularized* tactical architecture was included to test the utility of sensor IHU modules. In the formal definition of the architecture, sensors are directly connected to their associated IHU's, and actuator IHU's only receive the processed signal from the sensors generated by their associated IHU; never receiving a straight reading of the raw sensor value. In section 4.2.6, it was

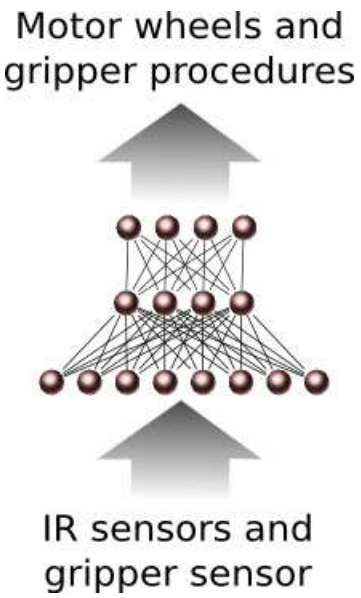


Figure 5.3: Architecture *(a)*: monolithic feed-forward architecture with no modularization.

shown that the use of a sensor IHU improved the fitness value. It would appear to be a sound assumption to think that having a dedicated neural network for a sensor may serve as a pre-processing element and assist in the robot control. It is not clear however whether the computational cost is worth the increased complexity level of the schematics. We will see how useful the sensor IHU's are on a medium size complexity example by comparing the results obtained with this architecture. As will be shown later, sensor IHU's proved very useful for robot control.

5.3 Comparison of results obtained

Figure 5.10 shows the evolution of the average fitness over generations for the different architectures. A complete list of the fitness evolution of each simulation can be found in appendix D. The results obtained with the first three architectures were very similar to those reported in [Nolfi, 1997], where those architectures were compared in the same task. Small differences related to the maximal fitness obtained may be appreciated, though not with regards to the behaviour of the architectures and their relation in performance. These differences may be due to either the use of a different evolutionary algorithm or to the minor details described in the 5.1 section.

From these results we can conclude that all the architectures evolved the correct behaviour. The only difference between them is the convergence speed and

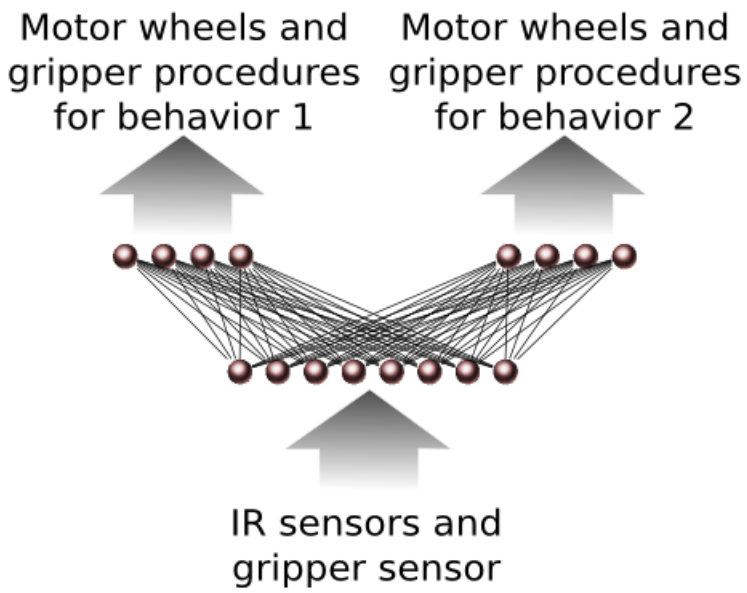


Figure 5.4: Architecture *(b)*: using distal strategic modularity.

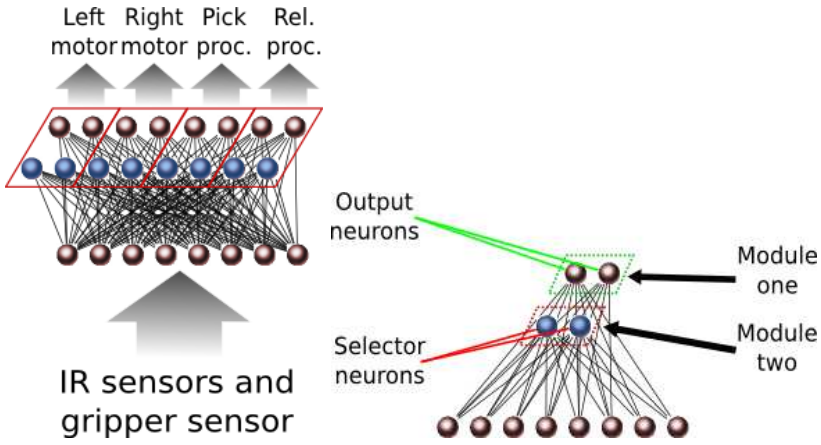


Figure 5.5: Left figure: architecture *(c)*, using proximal strategic modularity. Right figure: detail of the two modules that control one of the robot's actuators.

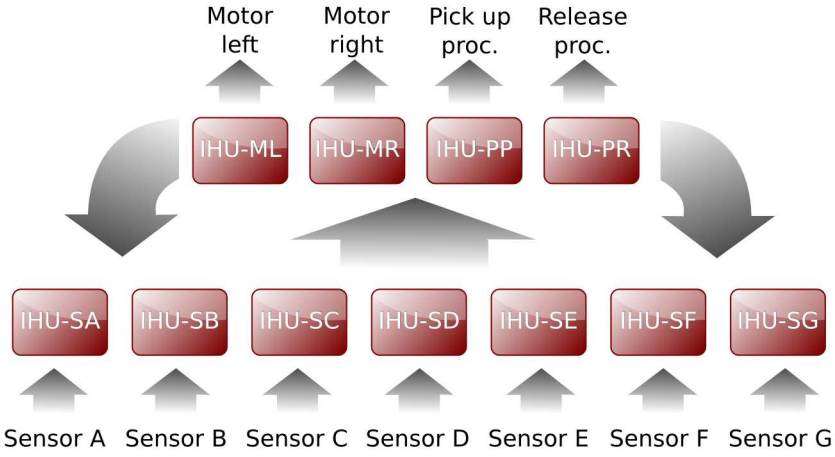


Figure 5.6: Architecture *(d)*: DAIR architecture using only tactical modularity. For the sake of simplicity the IHU's and their connections are represented by blocks. Arrows represent the interconnections between all the IHU's as was specified in the case of four IHU's in the simple example of section 4.2.6 (figure 4.9).

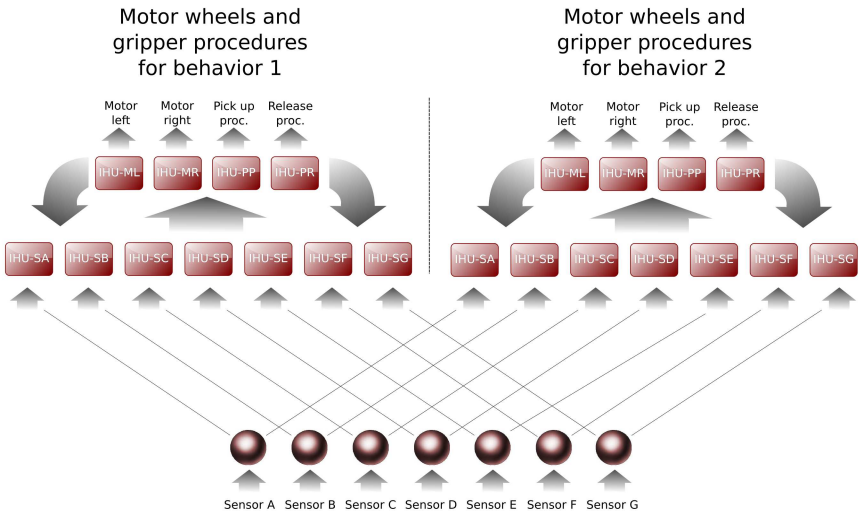


Figure 5.7: Architecture *(e)*: DAIR architecture using strategic + tactical approach.

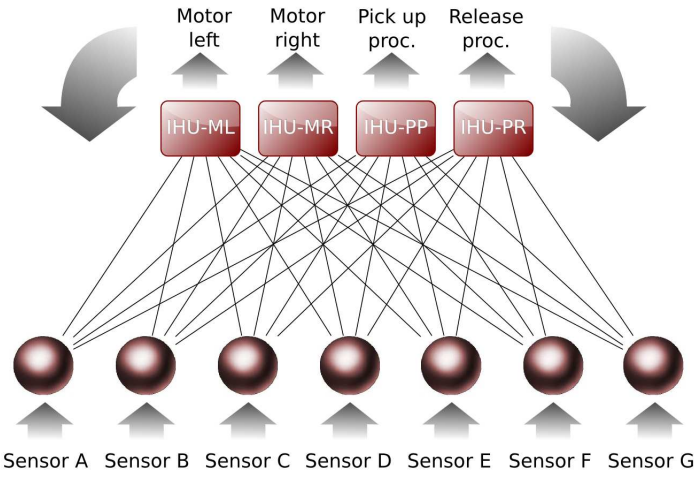


Figure 5.8: Architecture *(f)*: a tactical approach without sensor IHU's.

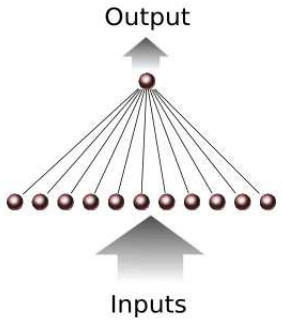


Figure 5.9: Neural network used in each IHU module of architectures *(d)*, *(e)* and *(f)*.

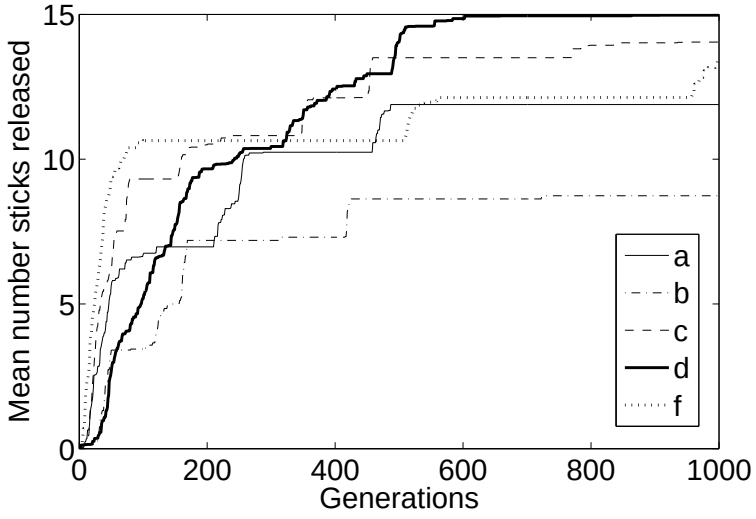


Figure 5.10: Number of epochs (out of 15) across generations in which individuals with different architectures (except architecture *(e)*) correctly picked up and released a target object outside the arena. Each curve represents the average of the best individuals in 10 different simulations.

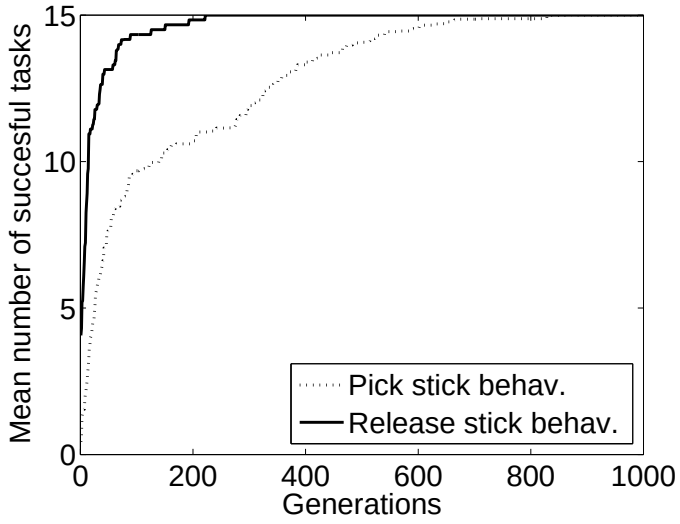


Figure 5.11: Number of epochs (out of 15) in which each of the architecture models *(e)* correctly performed their task (module 1 to pick up a stick, and module 2 to release the stick). Each curve represents the average of the best individuals in 10 different evolutionary processes.

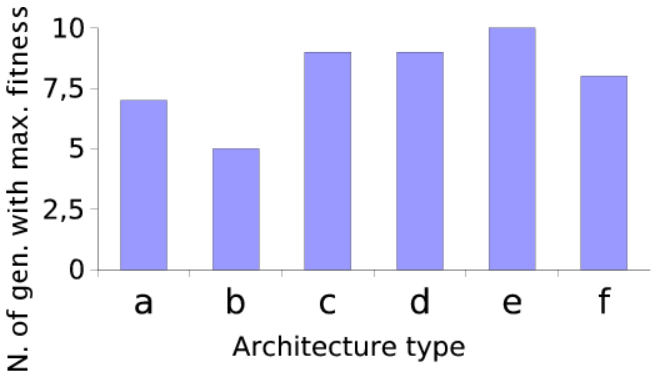


Figure 5.12: Number of times out of 10 that each architecture was able to reach maximum fitness.

the maximal fitness achieved, that is, the number of successful epochs (averaged over 10 evolutionary runs). Results obtained by the emergent architecture *(c)* and the two tactical modular ones *(d)* and *(f)* are very similar, and superior to the results achieved by the monolithic approach *(a)* or the strategic modular approach *(b)*. The tactical DAIR controller is the architecture that achieves the maximum fitness, - even though it takes longer than the others to obtain an acceptable behaviour (its evaluation speed is slower when compared with most of the architectures).

Within the tactical architectures, the semi-modularized architecture *(f)* performs below the completely modularized architecture *(d)*, indicating that the extra architecture modularization in *(d)* indeed performs a useful function. However, the *(f)* architecture achieves a fairly competent level of expertise substantially faster than *(d)* (it is the fastest architecture to achieve a fitness over 10). This may be due to the fact that the search space for the *(f)* architecture is smaller than for *(d)*, initially allowing for a quicker improvement (until it reaches a plateau). Additionally, the *(e)* architecture which combines both types of modularity is also able to master the task. Both of its modules evolved the maximum fitness very quickly (see figure 5.11).

Figure 5.12 shows a comparison between architectures indicating the number of times out of 10 that each architecture was able to reach the maximum fitness (15). Architecture *(e)* was considered to have reached maximum fitness only when both groups of tactical controllers did so. Again, results for the highly modularized architectures (namely, *(c)*, *(d)*, *(e)* and *(f)*) were better than those of the less modularized ones (architectures *(a)* and *(b)*).

Additional test

An additional test was introduced in order to test the *robustness* and real performance of the solutions. Robustness is understood as the capacity to generate a controller to solve the garbage collector task in the greatest number of different

(random) situations. The use of a performance metric permits us to measure the architectures not only in terms of fitness but also in terms of the behaviour obtained. It provides a kind of measuring system of the behaviour obtained that goes beyond mere maximum fitness score, and measures the general behaviour obtained in a much broader context.

The test consisted of adding an additional step to the evolutionary process; whenever the controller being evaluated was able to remove one stick from the arena in each of its 15 epochs flawlessly, an additional game started. This extra game consisted of performing a complete round where the robot had to take the five sticks out of the arena without making any errors (as described above, an error consisted of crashing into a wall, releasing a stick inside the arena or on top of another stick, or trying to grasp a wall). If the extra game ended correctly with all five sticks released within the predetermined timeframe (set to 2000 time steps) the fitness for that phenotype was increased by 10 units, and another extra game was started. The starting of these extra games continued until the robot made an error during the game, or it was not able to extract the five sticks within the 2000 time steps. As the number of extra games solved increases, the robustness of the controller increases correspondingly, since it means that the controller is able to solve the problem in more different situations.

This extra game allowed the evolution of more robust controllers; performing the whole task of collecting the five sticks in more diverse conditions. Fitness evolution with this extra game is shown in figure 5.13. In this case, the emergent modular architecture was the most robust, achieving the highest number of consecutive correct games. It was followed by the two tactical architectures *(d)* and *(f)*. Additional results depicted in figure 5.14 show the averaged maximum number of times that each architecture ended a game before an error occurred.

There was no significant difference between the highly modular architectures *(c)*, *(d)* and *(e)*, but it should be pointed out that in terms of robustness the emergent modular architecture achieved the best results.

5.4 Discussion

By comparing the different architectures, it can deduced that highly modular architectures *(c)*, *(d)* and *(e)* outperformed all the other architectures in all aspects. It can be seen that tactically modular approaches produce similar results to those achieved with emergent modular architecture. Tactical modularity outperforms emergent modularity in terms of average fitness, and obtains the maximum fitness value the same number of times, while the emergent modular architecture correctly completes more games. From these results, it can be concluded that the differences between the two architectures are not very significant. Larger differences exist between the three highly modular architectures, the monolithic neural network and the distal-strategic ones in all aspects: evolution speed, maximal fitness attained and the robustness of the solution found. Hence, the first conclusion that can be drawn from the results is that architectures with a higher level of modularity perform better than less mod-

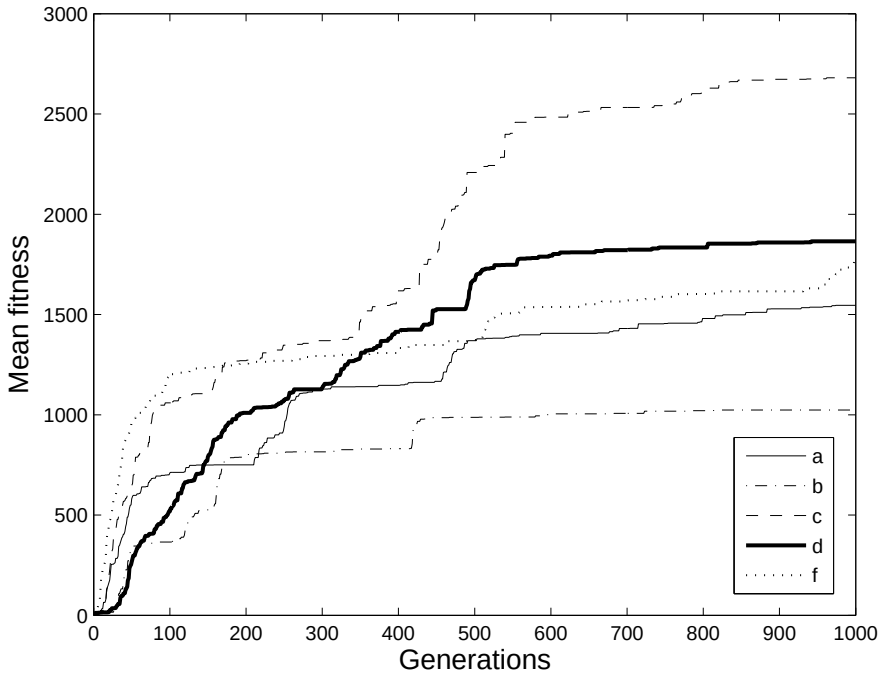


Figure 5.13: Evolution of fitness when the extra game with five sticks is added. Results averaged out over 10 evolutionary runs.

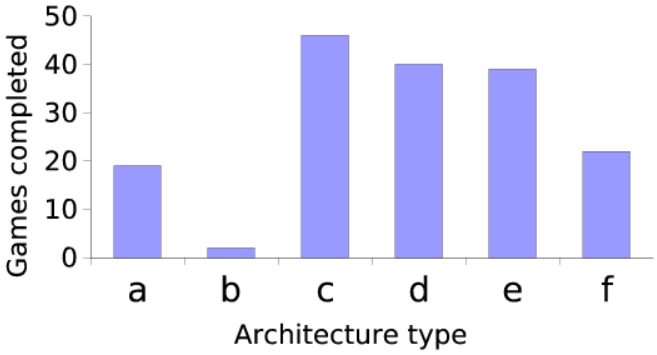


Figure 5.14: Maximum consecutive number of times that each architecture completed a game of releasing the five sticks outside the arena without making an error.

ularized ones. This result is in line with the main idea used in this thesis, by which a modular approach behaves better than a non-modularized one, and that modularity is required for complex behaviours.

Differences between emergent and strategic architectures exist at the conceptual level. In the case of the emergent modular architecture, modularity is performed in a strategic manner. The architecture requires the number of modules available for the evolutionary process to be specified beforehand. In the case presented here the architecture was provided with 2 modules; their optimal combination for the behaviour required was determined by the evolutionary algorithm. The complexity of the behaviour to be evolved must lay within the range given by the number of modules available for combination. Tactical modularity on the other hand does not require this module allocation, and the complexity of the behaviour generated is only limited by the complexity of the ANN used as processing element. In tactical modularity the evolutionary algorithm does not evolve the correct combination of modules for the behaviour required, but rather evolves an internal categorization of the behaviour, that is, it identifies which sub-behaviours are required for the target behaviour from a proximal point of view.

A strange result is obtained with architecture *(b)*. Even if its modularization degree is higher than architecture *(a)*, its results are poorer and far beyond the results of architecture *(a)*. The explanation to this effect may be explained by the problem of *genetic linkage* [Calabretta et al., 2003].

Genetic linkage affects evolutionary processes where two or more behaviours of differing complexities are required, and the ability of the evolutionary process to evolve the easier behaviours first prevents the evolution of the more difficult behaviours later. What happens here is that since both modules are being evolved at the same time, one of the tasks to be solved is simpler to evolve than the other, which means that the fitness increases by evolving the genotypes that better solve the easier task. Once the genotype has reached the maximum fitness possible for that task, the fitness should improve by evolving the genotype towards solving the other more complex task, but by that point the genotype is so biased in the fitness landscape that it cannot possibly modify the genotype to accommodate the other task without losing performance in the other. This problem is due to the impossibility of evolving both tasks at the same time. In [Calabretta et al., 2003] the effects of genetic linkage are shown applied to the *what and where* task [Jacobs et al., 1991a], a similar problem in terms of modularity to the garbage collector. A solution was proposed based on sexual reproduction which partially solves this problem.

The problem of genetic linkage between modules can be completely avoided if separated evolutionary processes for each module are used. This is the case for architecture *(e)*, where two different modules are evolved in different tasks using different evolutionary processes. In this case, two strategic modules were decided on from a distal point of view, and each strategic module was implemented by tactical modularity. The evolution of each strategic module is done in separated evolutionary processes, each one with its own fitness function. This mechanism allowed the controller to master each task separately and then combine them

into a single controller solution.

It must be recognized that sometimes it is not that clear as to how to divide a global behaviour into different simpler behaviours and perform different evolutionary processes for each one, as have been done here for the garbage collector. This is in fact one of the problems still to be solved in behaviour-based robotics. For these cases, the use of tactical modularity only is proposed (architecture *(d)*), which has been shown to be a lot better than strategic modularity alone (architecture *(b)*). For those cases, tactical modularity can improve the behaviour obtained especially if the robot is complex. Additionally, in the case of complex robots and complex tasks, tactical modularity offers the possibility of performing a progressive design of the controller, as will be explained in chapter 6, where a tactical-only controller is evolved for a complex robot.

In conclusion, from the results obtained it can be deduced that, generally speaking, highly modular architectures perform better than less modularized ones; and, even if it is possible to generate a monolithic controller for behaviour generation (as in architecture *(a)*), a highly modular approach is preferable if maximum performance is desired.

5.5 Additional results with a complex robot

In an additional set of experiments, the performance of the architectures was compared by using the complex robot Aibo. For these experiments, only the three most significant architectures were compared when evolving a controller for the Aibo stand up behaviour defined in section 4.3.4. In this task, the controller should learn how to stand the Aibo robot up from a lying position. The comparison was performed based on maximal fitness, fitness speed, and final pose achieved by the three different controllers. Architectures tested were the monolithic one *(a)*, the emergent modular *(c)* and the DAIR tactical *(d)*. Fitness function and ESP parameters used were the same as in section 4.3. Neural networks for each architecture were created in the following way:

- For the *(a)* architecture, the controller was based on a feed-forward network composed of 19 inputs corresponding to the 12 joint sensors, 4 paw sensors, and 3 accelerometer sensors. The network contained a hidden layer of 12 units. Finally, 12 output neurons were required for the control of the 12 joint motors. The algorithm thus evolved a total of 392 neural connections.
- For the *(c)* architecture, the emergent neural controller was designed with 19 inputs, for the same input as in the *(a)* architecture and two groups of 24 output units for the control of the 12 joint motors, according to the way the emergent modular works (see figure 5.5). The algorithm evolved a total of 456 neural connections in this case.
- For *(d)* architecture, the controller relied on 31 connected DAIR tactical modules with 31 inputs, 8 hidden units, and 1 output each. The algorithm was evolving a total of 7936 neural connections.

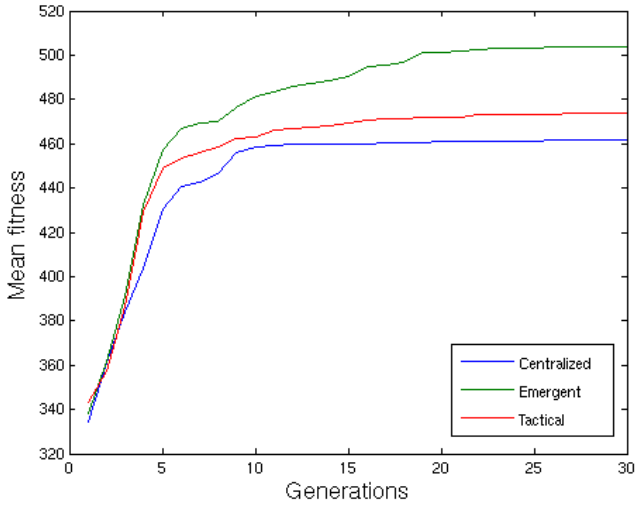


Figure 5.15: Comparison of the averaged fitness evolution for the three architectures (centralized, emergent modular and DAIR tactical).

Figure 5.16: Final positions obtained by the best controller obtained in the ten evolutions of each architecture. From left to right, architecture (a), (c) and (d).

The evolutionary process was performed ten times for each controller, and the fitness evolution was averaged for each architecture. Every evolutionary run lasted for 30 generations; long enough to obtain the desired behaviour.

5.5.1 Results

Figure 5.15 shows a comparison of the averaged fitness evolution for each architecture, and figure 5.16 shows the final position obtained for each of the three architectures.

Results obtained are similar to those in the case of the garbage collector. The three architectures performed the required behaviour. Maximum fitness was obtained by emergent modularity, with only a small difference to tactical modularity. Final pose is very similar in all three cases. These results indicate

that modular approaches obtain better results than monolithic ones. These results show how important and powerful modularity is, especially in a case like this, where the number of connections to evolve was quite different between the monolithic approach and the modular ones.

5.6 Conclusions

An exhaustive comparison between different modular controllers making use of these types of modularization in different combinations has been provided for the garbage collector problem, and the following conclusions can be listed:

- Modular architectures performed better than monolithic ones.
- The greater the degree of modularity, the better the fitness results obtained.
- Even if a monolithic controller can be evolved to perform a determined behaviour, a tactical modular version of that controller has performed better. Three different experiments have shown this result: the simple robot contour-following experiment in section 4.2.6, the Khepera garbage collector experiment in the 5.1 section, and the Aibo stand up experiment in the 5.5 section .
- A monolithic approach can be effective if no genetic linkage effect is produced (though hard to know before hand).
- A strategic modularization in separated evolutionary processes avoided genetic linkage in previous works. The DAIR approach proposes this method as the best one to create complex behaviours.
- Tactical modularity can either be combined with strategic modularity or used alone. Any combination has outperformed less modularized approaches.
- If a strategic evolution of two or more modules in separated processes is not possible, a simple tactical modular controller can improve the controller obtained by other approaches.
- Distributed modularization at the sensor level has helped to improve the fitness of a tactical modular controller.

Among the advantages of using a DAIR architecture in evolutionary robots stated in this chapter, the architecture presents two other interesting features that could become important in the generation of complex evolutionary robotics; namely *progressive design* and *internal representation*. The former will be explained in the next chapter, and the latter in chapter 8.

Progressive Design Software builds and delivers software designed to meet your needs completely while slashing the time it takes to achieve Return On Investment

Progressive Design Software company advertising

6

Progressive design through staged evolution

The DAIR architecture was introduced in chapter 4, and was applied to the control of both a simple wheeled robot and a complex legged one; however, the task evolved in all cases was a simple one. This chapter introduces one of the strengths of this architecture for the evolution of a complex coordination in a complex robot, namely *progressive design*. With the use of progressive design, the architecture minimizes the drawbacks of large search spaces and the bootstrap problem when evolving a controller for a robot whose body, task and environment are complex and determined beforehand. The main goal in progressive design is the generation of a tactical modular controller in stages. In the same manner as the DAIR architecture describes a modularization of the controller, the DAIR progressive design method provides a modularization of the learning procedure.

Basically, the progressive design process works as follows: at the beginning of the evolutionary process a limited number of modules are evolved in a bounded evaluation task. Following this the rest of the modules are added in successive stages, with only the newly added modules and their connections with the already evolved ones in previous stages being evolved. The final global controller is then gradually evolved in a process referred to as *progressive design of complex neural controllers*. The key point of this method is the level at which modularization of the neural controller was performed by the DAIR architecture definition. This new modularization level replaces the usual *functional modularization* (used in other approaches) with a *device modularization*, thus allowing a *customized* introduction of knowledge for each tactical module.

This chapter provides an in-depth description of the progressive design method for complex robots. A simple application of the method for the garbage controller problem of the previous chapter is first shown. Next, an application of this design to the complex Aibo robot is used for the generation of a walking gait.

6.1 Introduction

This chapter addresses the generation of complex sensorimotor coordinations for complex robots within the evolutionary robotics framework. By *complex sensorimotor coordinations* we understand controllers coordinating several sensors and actuators to generate a behaviour in several stages. The *complexity of a behaviour* will be defined based on the number of different stages that the evolutionary process will use to evolve such a behaviour. Of course the complexity of the behaviour will depend on the architecture adopted in the first place. This means that a given behaviour may present a different complexity for two different architectures. This fact may be useful to compare how architectures can decrease the complexity of a behaviour.

When facing complex controllers for complex robots in evolutionary robotics, it is preferable to insert as little human knowledge (otherwise known as *bias*) as possible in the evolutionary process. Bias determines to a certain extent the path that the evolutionary process must follow, thereby reducing the dimensionality of the evolutionary search. On the other hand, the inserted bias influences the evolutionary search towards a certain type of solution, limiting the amount of possible solutions that the evolutionary algorithm could potentially generate.

We however, claim that it is practically impossible to generate a controller under the conditions stated without introducing bias, as the artificial evolutionary algorithm is trying to reproduce the effects of natural evolution in unfair conditions. While natural evolution gradually evolved the animals body plan, their sensors and actuators, their nervous system and even their environment at the same time, artificial evolution tries to evolve the nervous system of a robot for a given morphology, group of sensors and actuators, and within a given complex environment. Even if this method has worked in some simple examples, as was shown in chapter 2, 4 and 5, no such satisfactory results exist for complex robots due to both the large search space that the algorithm has to face, and consequently, the bootstrap problem. And so, the principle that some constraints should be relaxed if a controller is going to be evolved for such complex robots is defended in this thesis. Since robot morphology, task and environment are fixed and complex, one possibility is to relax the bias constraint.

6.1.1 Similar works

The *evolving by stages* concept, being one of the clearest examples of the use of incremental evolution (see section 2.4.1) has already been used in literature. In incremental evolution, controller evolution follows a progressive complexification *of the task* to evolve. The evolutionary process begins with an easy-to-reach task which is related to the goal task. The designer then successively increases its complexity towards the goal task. In this case, bias information is introduced in the decision of the sequence of challenging tasks. This approach has in fact worked well in relatively complex tasks with simple robots [Islam et al., 2001], but successful use in complex robot has to date not been reported.

In [Urzelai et al., 1998, Dorigo and Colombetti, 1998] special attention is paid to the possibilities that a modular architecture for *gradual shaping* offers, that is, incremental training of independent behavioural competencies. The problem dealt with here centres on how much human design should be included in combination with learning algorithms in these processes to find a solution for the problem . As a consequence, the entire learning activity should be organized in the appropriate way. The BAT methodology (Behaviour Analysis and Training) was proposed. It consists of performing a rational organization of the main phases of the robot controller design, which are: application description, behavioural analysis, specification, training and behaviour assessment. In this process, the design and implementation of the robot's physical body as an important part of the process was also included. In the first step, the creation of a sufficiently detailed description of the robot, the environment and the target behaviour is laid out. In the second step, an in-depth analysis of the behaviour required is used to break such behaviour down into a set of simpler behaviours. The third step involves the creation of the sensorimotor structure necessary for that behaviour. The fourth step is about the implementation of the behaviour modules and their interconnections: either by hand or through an automatic mechanism. A training system must then be provided to train the modules. The final step is the assessment of the obtained controller in the task at hand, which could lead to a repetition of all the design steps if the results are not satisfactory.

The BAT work is interesting for the DAIR approach in the sense that it also stresses the importance of a careful analysis of the behaviour required and the knowledge already available for its generation; however, it also lacks an application for complex robots.

Other staged approaches to manually design a modular architecture are those whose modules are completely decided on by the designer by taking inspiration from biological systems. In such cases, designers decide which modules to use, how to construct them and how to connect them to each other. Some biological inspiration is certainly incorporated, but a lot of ad-hoc information is also used. An example of this methodology is [Ijspeert, 2001, Hallam and Ijspeert, 2003] for the generation of a walking and swimming lamprey. In this case, the concept of Central Pattern Generator (CPG) is used. A CPG is a group of neurons which generate as output an oscillatory pattern from a non-oscillatory input. It has been proven that CPG's are behind the walking mechanisms of some animals, and it has been further suggested that the same may be true for humans. It could be said that this work performs a *staged evolution*. In staged evolution, incremental evolution can be used not only to modify the fitness function at every stage but also the neural structure to evolve at those stages.

A different case of biological inspiration used for staged evolution is that in [Manoonpong et al., 2005], where we see the evolution of a modular neural controller for the control of a four legged reptilian shaped machine. The same CPG concept is used to drive robot motors. The control mechanism was complimented with a neural module for velocity control and another to process sensory inputs. However, many ad-hoc solutions were used in this example.

Other staged approaches use human intuition: [Lara et al., 2001] in the evolution of neural modules and their interfaces, or [Muthuraman et al., 2003, Muthuraman, 2005] in the evolution of modular controllers for the generation of walking robots. This latter system is aimed to be applied to the growth of any type of neural structure (walking control, vision control, and others). In [Doncieux and Meyer, 2004] bias information is used to help the evolutionary process to decide which of the controller's connections are likely to be useful. In [Nelson et al., 2003b], fitness functions with two (or more) different modes are used. The first mode rewards the controller when it has been incapable of completing the task, and uses a subjective measure (completely determined by the designer) of the uncompleted task, with the second mode only providing a reward when the task is achieved.

All the listed works use bias information in different evolutionary stages. The adaptation of the controller in different stages, that is, staged evolution, implies that there is a *modularization at the learning level* [Auda and Kamel, 1999], whether the neural structure is modular or nonmodular. On the other hand *modularization of the neural architecture* is an alternative. The complete DAIR architecture description incorporates both types of modularization at its very foundation. Hence, from the works described, for the evolution of a neural controller for a complex robot, the process should:

- Use external information
- Break the controller down into modules
- Break the learning process down into stages

The rest of the chapter will show how the DAIR architecture is well situated to implement those three requirements, and how it is best suited for the selective introduction of external information in selected modules.

6.2 Progressive design of neural controllers

In previous chapters, tactical modularization has been used to generate controllers when the behaviours evolved were *simple*. This means that most of them only required a single stage to be evolved. However, a complex behaviour could be required, not even a tactical modularization is able to obtain a suitable controller. For this reason, a learning mechanism has been designed which implements staged learning, with bias information playing a primary role. The DAIR approach includes the use of the concept of modularity in learning together with modularization in architecture as a solution to the evolution of complex controllers. In fact, as we will see below, our tactical modularization approach presents significant benefits for the use of modular learning.

As well as how to modularize a neural architecture in order to obtain a fitness improvement having been introduced in previous chapters, in this chapter how to modularize the learning process using our modular architecture will be shown. This procedure is called *progressive design*.

6.2.1 Description

This section describes our *progressive design* method to minimize the effects of large search spaces and the bootstrap problem when evolving a controller for a robot whose morphology, task and environment are fixed and complex. It is based on staged evolution, and extends it by providing a standard building block and evolutionary method for any robot, independent of its number of sensors or actuators. In the progressive design method, architectural modularity is created at the robot device level by creating a small and independent neural module around each of the robot's sensors and actuators. This change in modularity has in chapter 5 been proven to be more effective than other modularization approaches, and it additionally adds the advantage of allowing a very flexible learning modularization.

In the progressive design approach, tactical controller evolution is performed in several stages, as follows:

Setup. Let's assume we have a complex robot with a huge number of sensors, M, and actuators, N. The vector of perceptions $\mathbf{y}$ is defined as the vector containing all the values for the sensors,

$$\mathbf{y} = \{y_k\}_{k=1}^{M}$$

The vector of actions $\mathbf{u}$ is aso defined as the vector containing all the commands generated for the actuators,

$$\mathbf{u} = \{u_k\}_{k=1}^{N}$$

Therefore, the input size of each ANN module, - for either a sensor or an actuator, will be equal to the number of modules, that is:

$$num.inputs = size\{\mathbf{y}\} + size\{\mathbf{u}\} = M + N$$

For this robot, a goal-task t_g is given which is associated to the behaviour required for the robot. The goal is then to build a distributed DAIR controller composed of $M+N$ modules, which achieves t_g.

First stage. A subset of the sensors $m \leq M$, and actuators $n \leq N$ of the robot is selected by the designer depending on the *evolutionary strategy* to be performed. This is the first place where bias is introduced, since the designer decides which and how many devices are used in this stage. Now, a tactical modular controller for each of the $m + n$ selected devices is designed using the method described in chapter 4; by assigning one IHU module to each selected robot device. A simple evaluation task t_1 is also designed to evolve this group of modules, related to the final goal t_g in some sense, and relevant to the selected devices. Since the required task is decided on by the designer, this is the second point where bias is introduced. Task t_1 is associated with a fitness function f_1. Hence,

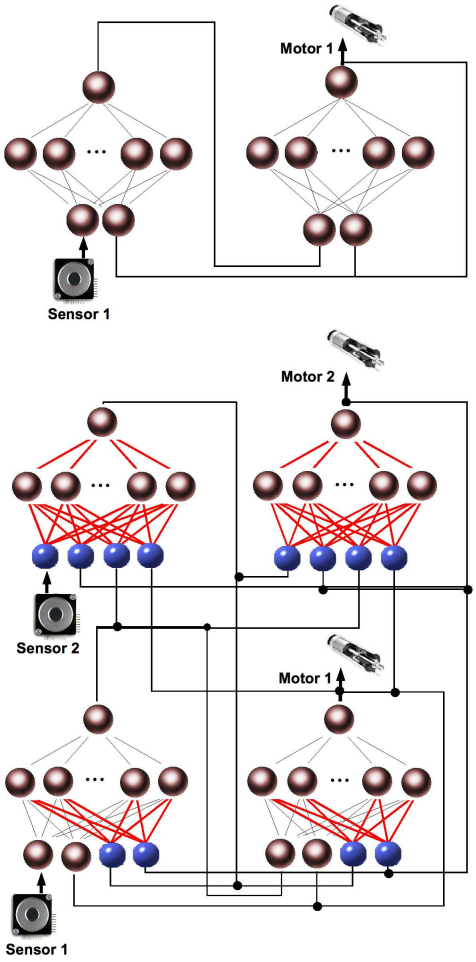


Figure 6.1: The first example of the progressive design of a neural controller for a simple robot with two sensors and two actuators. In stage one (top), only one IHU for sensor 1, and one IHU for motor 1 are evolved, using a given fitness function f_1, and obtaining the controller on the top. In stage two (bottom), two additional IHU's (for sensor 2 and motor 2) are added to the controller. At this stage, only those newly added modules and their connections to the previously evolved modules (red lines) are evolved by using a different fitness function f_2. Bias information is used to decide which IHU's are going to be evolved first, with which combination and using which fitness function. Hence, the designer requires knowledge of the domain .

the neural modules of those sensors and actuators selected are evolved together in the simple task t_1 using fitness function f_1.

$$\mathbf{y}_{t1} = \{y_k\}_{k=1}^{m} \ , \ \mathbf{u}_{t1} = \{u_k\}_{k=1}^{n}$$

$$num.inputs_{t1} = size\{\mathbf{y}_{t1}\} + size\{\mathbf{u}_{t1}\} = m + n \leq M + N$$

Iteration. Once the small group of $m + n$ controllers has gained proficiency in their evaluation task t_1, the values of their connections are frozen and a new evolutionary stage begins. For this new stage, a new and different sub-group of the robot's m' sensors and n' actuators is selected for the evolution of their modules. Their neural modules are added to the previously evolved group of modules, as well as interconnections between the modules of the first evolution and the new ones. The weight values of the connections with the old ones (in dotted lines in figure 6.1) are initially started with small values, resulting in a slight modification of the already evolved behaviour in the previous stage. Now the sensor and actuator vectors of the DAIR controller have increased in size as follows:

$$\mathbf{y}_{t2} = \{y_k\}_{k=1}^{m+m'} \ , \ \mathbf{u}_{t2} = \{u_k\}_{k=1}^{n+n'}$$

$$num.inputs_{t2} = size\{\mathbf{y}_{t2}\} + size\{\mathbf{u}_{t2}\} = (m + m') + (n + n') \leq M + N$$

The evolutionary process is started again, but only for the newly added modules and their connections to the modules of the previous stage (red lines in figure 6.1). For this new stage, the evaluation-task is changed to a new task t_2 that may be different from the previous t_1, although this is not mandatory. It is important to note that the new task t_2 deals with the whole group of modules, even if only the newly added ones and their connections with the modules of the t_1 task are evolved. The evolution will continue until the group of devices gains enough proficiency in task t_2. Once a certain level of proficiency in task t_2 is reached, a new stage starts, where more sensors and actuators are added, and the evolutionary process is iterated until the final number of $M+N$ modules is reached. More and more of the robot's sensors and actuators are progressively added, and the evaluation task they perform modified. Eventually, the total number of sensors and actuators should be reached, and the final global task required should be mastered by the controller.

6.2.2 Discussion

The idea behind progressive design is as follows: start evolving a *subset of modules* in a *related evaluation-task*, and then *progressively increase* the number of modules that are evolved in related evaluation-tasks. The key point is that the evaluation-task is only related to that subset of modules that is being evolved at that particular stage. By doing this, the subset of modules establishes the coordination required between each other for the resolution of their associated evaluation-task. When the new modules are added in a second stage for the

evolution of a second evaluation-task, the new modules will need to learn the newly assigned evaluation-task, but at the same time they will (slightly) modify the already existing evaluation-task evolved in the previous stage (by means of the new connections with the old modules). When this procedure is repeated through all the required stages and modules the final controller is gradually shaped into the desired controller.

On this first approximation to the progressive design method, the addition of new modules to an existing controller implies the evolution of new input weights in the previously evolved modules. These connections mean the effect of the newly added modules over the old ones. However, it is not guaranteed that this direct connection will allow the evolution of a coupled controller that performs the required behaviour, in fact, what is happening is that the influence on the old modules is a simple linear combination output of the newly added IHU's. A more complex influence may be required; for instance, additional connection neurons may be used that act as a coupling between groups, as shown in [Lara et al., 2001]. The addition of such coupling neurons would allow smoother and more complex interactions between different groups of modules. These more complex solutions have not been explored here, and their study is left for the future.

The evolution of modules in stages allows for the minimization of the drawbacks of the two main problems of evolutionary robotics previously described, namely a large search space, and the bootstrap problem. First, by evolving a limited number of modules at each evolutionary stage of a simplified task the searching space for the evolutionary algorithm is kept narrowed. Secondly, by evolving new modules keeping the functionality obtained in previous stages, the likelihood of encountering a bootstrap problem is reduced, since the evolution of the newly added modules depart from a previous stable solution which can be scored, and thereby drive the evolutionary path from the initial generations. This effect will be clearly seen in the example shown in section 6.3.

On top of that, there is an additional advantage of using this method. Due to the modularity of the controller performed at the device level it is possible to easily insert external knowledge for the evolution of any device or combination of them: separated groups of modules can be trained in different tasks, and then be merged into a single DAIR controller simply by evolving connections between them; the evolved group of modules can be reused by performing a copy of the controller evolved for a group of devices to control another group of devices which are exactly the same (e.g., modules controlling a robot right arm can be duplicated to control the left arm). This architecture allows a powerful use of external knowledge, because knowledge will affect only those elements concerned.

As indicated in [Urzelai et al., 1998, Dorigo and Colombetti, 1998], adaptation in modular architectures can take place at two different levels: within a module and between modules. It is clear that our architecture acts at both levels; within a module, since each tactical module is adapted to control its associated device, and between modules, because connections between tactical modules are carefully adapted for the generation of a coordinated behaviour.

In fact, when symmetry appears in a problem, the t_k task can be evolved for one single group sensor-actuator, architectural results can be copied to the symmetric group, and then evolve the upper task t_{k+1} by evolving the connections between both groups (as depicted in figure 6.2). Either way, it is up to the designer to decide the best schema for the given task, based on the available information.

6.3 Application to Khepera garbage collector

This section will show an application example of the progressive design methodology for a simple robot, - the Khepera garbage collector, in order to illustrate how it works. The selected test bed follows the description provided in the previous chapter. It will be assumed that it is not possible to evolve the required behaviour for the robot in one single evolutionary round (as we know, this is not true, since it has already been obtained in chapter 5). An application to a complex robot in a complex task will follow in the next section.

6.3.1 Experiment setup

Experimentation will follow the description of the same setup in chapter 5. Basically, experiments consisted of 15 epochs of 200 time steps each, where an evolved controller was tested in the task. The duration of each time step is 100 ms. Epochs end after 200 time steps or after a stick had been correctly released outside the arena.

The final goal is the generation of a tactical modular controller for the Khepera robot to solve the garbage collector problem. The controller will be composed of eleven modules: six infra-red sensors, one gripper sensor, two motors and two gripper procedures. Each neural module was implemented using a feed-forward neural net with no hidden units and only one output. At the beginning of the progressive design of the controller, the number of inputs of those nets will depend on the number of elements selected for the first evolutionary stage. In any case, at the end of the progressive design the neural net for each module will have eleven inputs.

6.3.2 Progressive design strategy

The architecture was evolved using the progressive design process based on a three stages procedure. In the first stage, not all of the sensors were used to reduce the number of weights to be evolved. Only seven modules were evolved: three sensors and all four of the actuators. In successive stages, new sensors were added increasing the proficiency of the robot in the task, and hence the fitness obtained.

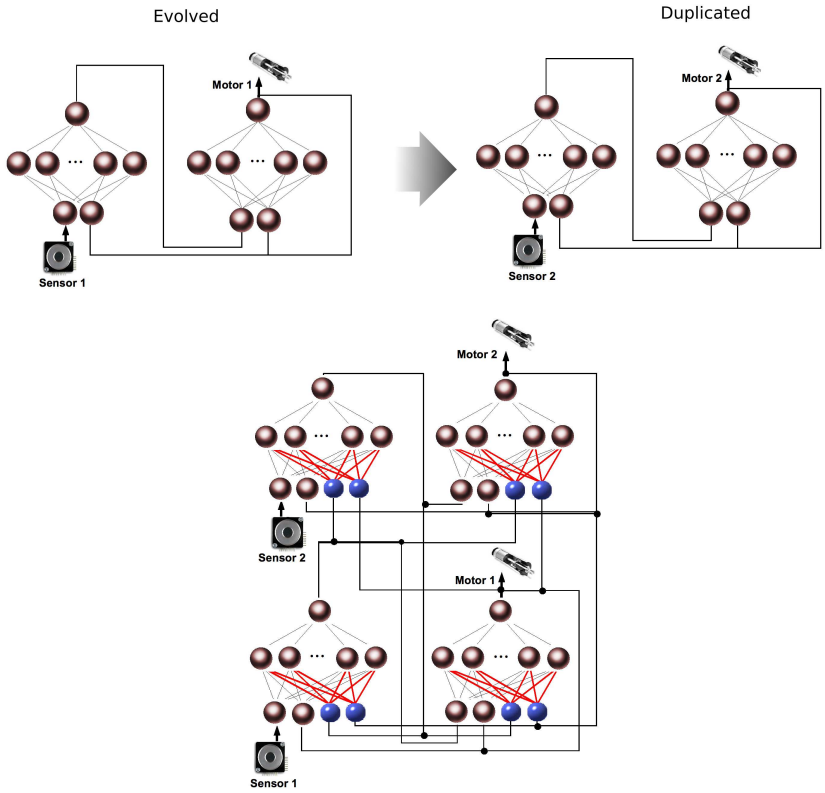


Figure 6.2: Second example of the progressive design of a neural controller for a simple robot with two sensors and two actuators. Available information about the task to evolve indicates that it is symmetrical, i.e., the job performed by the sensor-1 and motor-1 (for instance, a right arm) should be the same as the job performed by sensor-2 and motor-2 (a left arm). Hence, in the first stage, a group of *one sensor-one actuator* is separately evolved on task t_1. The obtained controller is then duplicated for the control of sensor-2 and motor-2. In the second stage only the connections between both groups are evolved in the final task t_g (connections in red line) obtaining the final controller.

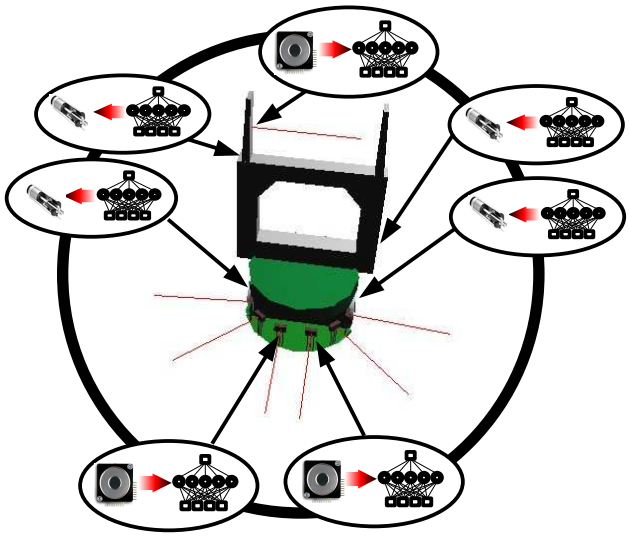


Figure 6.3: First stage for the evolutionary controller: only seven modules were evolved, corresponding to three sensors and four actuators.

6.3.3 First evolutionary stage

In the first stage, evolution was started using as few sensors and actuators as possible in order to maintain a small initial search space, as well as the robot performing at some low degree the garbage collector task. It was decided that the two front IR sensors were to be used as a minimum to discriminate between sticks and walls. For grasping and releasing sticks the gripper sensor is required, as well as the two wheel motors and *take* and *release* procedures, a total of seven IHU's (figure 6.3),

$$\mathbf{y}_{\mathrm{t1}} = \{y_{Sc}, y_{Sd}, y_{Sg}\} \ , \ \mathbf{u}_{\mathrm{t1}} = \{u_{Ml}, u_{Mr}, u_{Pt}, u_{Pr}\}$$

A fitness function was designed for the evolutionary process to reward the controllers for releasing a stick outside the arena. An additional term was added which rewarded robust controllers, that is, controllers that performed the stick-releasing behaviour without performing any errors; errors included crashing into walls, releasing sticks inside the arena or on top of another stick, or trying to pick up a wall. Controllers that were able to pick up only one stick were also rewarded with a lower fitness value.

$$fitness = \begin{cases} 1 & if\ pick\ up\ stick \\ 100 & if\ stick\ released\ outside\ arena \\ 110 & if\ stick\ released\ without\ errors \\ 0 & otherwise \end{cases} \tag{6.1}$$

Similar to experiments in chapter 5, a special mechanism was implemented which artificially added a stick in front of the robot each time it picked a stick

Figure 6.4: Genotype encoding for the first stage of the Kephera garbage collector.

up. Each evolutionary process lasted for 1000 generations. Each genotype was encoded according to the schema in figure 6.4.

After 1000 generations, the maximum fitness obtained was 1531 out of 1650. The behaviour obtained, even though sub-optimal, permitted the robot to solve the task. Due to the stochasticity of the method employed the evolutionary process was performed ten times, obtaining a mean fitness value of 1021. Eight out of ten evolutionary processes were able to generate the garbage collector behaviour within 1000 generations. Figure 6.9 shows the evolution of the mean fitness value over generations.

6.3.4 Second evolutionary stage

The second stage began by selecting the best group of modules evolved in the previous stage from the ten evolutionary runs performed. These modules were frozen in their evolution, that is, they were not evolved any further. We refer to them as the S1-modules or S1-controller. Next, two new sensor IHU's were added to the controller; namely, the ones corresponding to the control of the two diagonal IR sensors (sensors B and E as defined in figure 5.2),

$$\mathbf{y}_{\mathrm{t2}} = \{y_{Sb}, y_{Se}, y_{Sc}, y_{Sd}, y_{Sg}\}\ ,\ \mathbf{u}_{\mathrm{t2}} = \{u_{Ml}, u_{Mr}, u_{Pt}, u_{Pr}\}$$

During this stage, only the weights of these two new modules and their connections to the already existing S1-modules were evolved. The genotype for this stage, that is, what was actually evolved, was encoded as indicated in figure 6.6.

New connections between the output of the new modules and the inputs of the old modules were randomly initialised by the evolutionary algorithm with very low values of between -0.001 and 0.001, and evolved in separated sub-populations. Each sub-population contained the new connections from all the new modules to one old input.

For this particular task and combination of sensors and actuators, the fitness function used in this stage is the same as in the previous one, as defined in the 6.1 equation. For other cases, the fitness function can change at this point if necessary in order to implement a different task, starting from the knowledge of the task learnt in the previous stage. This is not the case for the garbage collector problem, and the same fitness function can be used even if the structure

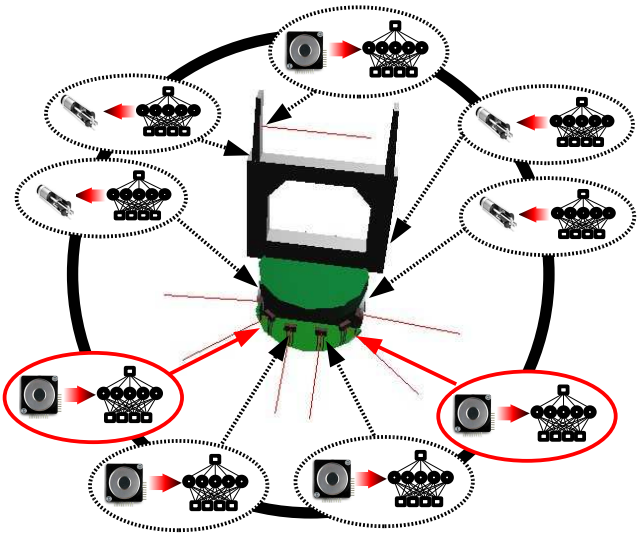


Figure 6.5: Two additional sensor modules were added at the second stage (in red).

Figure 6.6: Genotype for the second stage of the Kephera garbage collector.

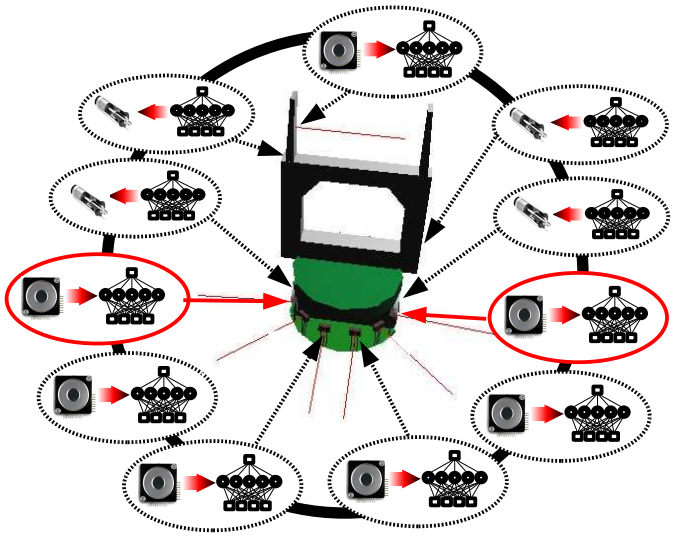


Figure 6.7: Two additional sensors were included at the final stage, (in red).

of the controller has changed. An example of a task requiring the fitness function to change at each stage will be discussed in section 6.4.

Figure 6.9 shows the evolution of the mean fitness value over generations for this stage. Looking at the fitness evolution, it can be observed that at the beginning of this new evolutionary stage the fitness obtained at the very first generation is lower than the maximum obtained in the previous stage due to the interference that the newly added connections are producing in the solution found in the previous stage. However, the fitness of this first stage generation is not low, since the neural weights obtained in the previous stage represent a solution point in the fitness landscape near to the final solution. After 1000 generations the maximum fitness reached was 1650 out of 1650. The evolutionary process was performed ten times, obtaining a mean fitness value of 1570 after 1000 generations. All ten evolutionary processes were able to generate the garbage collector behaviour.

6.3.5 Third evolutionary stage

At the third stage, the IHU's for the two lateral sensors were introduced (sensors A and F, see figure 6.7). The same procedure as in the previous stage was used. The best controller evolved in stage-2, which contained modules from stage-1, was selected and its evolution frozen,

$$\mathbf{y}_{\mathrm{t3}} = \{y_{Sa}, y_{Sf} y_{Sb}, y_{Se}, y_{Sc}, y_{Sd}, y_{Sg}\} \ , \ \mathbf{u}_{\mathrm{t3}} = \{u_{Ml}, u_{Mr}, u_{Pt}, u_{Pr}\}$$

Only the two newly added modules and their connections to the already existing ones were evolved. Again, the fitness function defined in the 6.1 equation was used for this stage. The genotype was encoded as illustrated in figure 6.8.

Figure 6.8: Genotype for the third stage of the garbage collector.

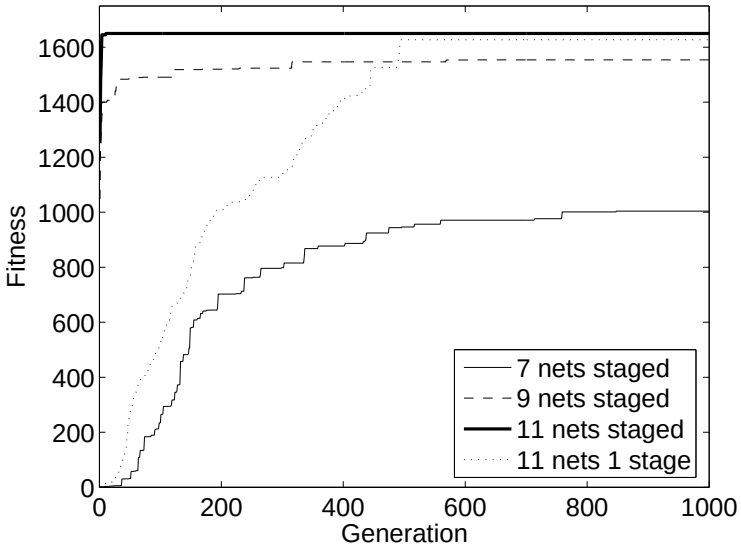


Figure 6.9: Mean fitness evolution through generations for all the stages. The dotted line shows the fitness evolution obtained when the eleven modules are evolved in one single stage (result obtained in chapter 5). Values have been averaged over ten evolutionary runs in all cases.

For this stage, the fitness evolution had the same behaviour as in the second stage, that is, the fitness obtained in the first generation of this stage is a slightly reduced value from the maximum obtained at the end of the previous stage. However, at this stage, the reduction is smaller than the reduction experienced at stage two. Our hypothesis is that the solution found at the previous stage is very related to the solution with two more sensors. In fact, the task is the same but uses two more sensors, so that the solution only has to accommodate the action of the newly added sensors.

After 1000 generations the maximum fitness value is 1650 out of 1650. The evolutionary process, performed ten times, leads to obtaining a mean fitness value of 1647 after 1000 generations. All ten evolutionary processes were able to generate the garbage collector behaviour. Figure 6.9 shows the evolution of the mean fitness value over generations.

6.3.6 Comparison with one single-stage evolution

The previous chapter demonstrated that it is possible to obtain the garbage collector behaviour for the eleven modules in one single stage. This is due to several facts, which include the low complexity of the robot used, the simplicity of the task to be solved, and the power of the architecture and evolutionary algorithm. If the case of one single-stage evolution is compared with the progressive design approach, it can be observed that the mean fitness value of the former is 1645, a fitness slightly below the mean fitness obtained by progressive design (1647). Figure 6.9 shows the evolution of the mean fitness value over generations (dotted line), compared with the evolution of the fitness of the other stages.

Furthermore, in the case of single stage evolution, nine out of ten evolutionary processes were able to evolve a maximum fitness garbage collector behaviour, while in the case of the progressive design all of the controllers from the last stage were able to generate the garbage collector behaviour (10 out of 10). Even though both approaches obtained similar results, a small improvement has been attained of the progressive design in terms of maximal mean fitness achieved and number of times that the behaviour evolves. This results seem reasonable, since in the progressive case the controller is carefully built step by step starting from previously known domains, which allows for a smoother evolutionary process.

However, the improvement in fitness obtained may not be worth the complexification of the evolutionary process. This complex procedure may only be required in more complex situations where the single-stage evolution is not possible. This is the situation depicted in the next section.

6.4 Aibo walking

The results presented in the previous section show how the methodology works for a simple case. However, the main aim of this methodology is its use in complex robots. This section shows results obtained when the methodology is applied to the Aibo robot for the generation of a walking behaviour. The generation of such a behaviour is a complex issue, since Aibo endows 12 DOF's related to walking, with no head or queue DOF being taken into account. Each leg has 3 DOF's that must be coordinated between all the legs. It is easy to see that a miscoordination in one single joint may cause the robot to fall. In fact, there is no work to date in literature that shows a straight evolution of a neural controller for a walking behaviour for such a robot. By straight evolution we refer to a single controller evolved using either a single fitness function or an incremental one. All previous works on this subject evolved separated parts in different processes and then attached them, as we will see below.

For the generation of the walking gait only the motors in the leg joints and those corresponding joints sensors will be taken into account. Neither the paw sensors nor the accelerometers will be used in this case since it seems unnecessary

for a walking gait. The whole process will be carried out in simulation, and once finished it will be transferred to the real robot and tested on it[1].

It must be stated that the goal of this section is not to show how a walking behaviour must be performed in a quadruped robot. Generating a walking gait for a quadruped in a highly complex robot with several DOF's is a rather complex task, and has been solved using non-linear dynamic CPG equations in several studies such as [Lewis et al., 1992, Collins and Richmond, 1994, Reeve, 1999, Lewis, 2002, Markelic, 2005]. The primary goal is to show how such a complex behaviour for such a complex robot can be evolved using the progressive design method. Furthermore, the walking gait presented here can be optimised for energy consumption, velocity or aesthetics. It is also stressed that the progressive design method is a general methodology for use in the evolution of any robot and/or behaviour. This thesis shows just two possible applications (the garbage collector for a Khepera robot and the walking for an Aibo robot).

6.4.1 Technical issues: joint control

A typical method for the control of a joint position is by indicating what the position of the joint should be at each time step. This method, referred to as *control by position*, is the most commonly used in most trajectory definition methods for walking robots. The trajectory to be applied has been calculated or designed beforehand, and once it is acknowledged as correct it is applied to the robot.

Another type of control is possible, known as *control by torque* or otherwise stated *control by velocity*. In this type of control, the control signal indicates the torque that the joint has to apply at a given moment, which results in a velocity in the joint. The Webots simulator implements both types of controls, and it is therefore up to the user to decide which one to use. However, due to the inner workings of the simulator, only velocity control will be possible for this experiment:

in Webots, the mechanism underlying the control of a joint is composed of three differentiated phases which work as follows [Cyberbotics, 2005]: every joint has a defined maximum speed and a maximum acceleration. Once a given position for a joint is ordered, the joint then starts accelerating at the maximum acceleration rate from zero velocity to the maximum velocity of the joint. Once the maximum velocity is reached the joint remains moving at that velocity. Just before reaching the required position for the joint, the joint starts decelerating at the maximum deceleration rate. At the end of the process, the joint achieves the required position with zero velocity. This process has been represented in figure 6.10. If the correct joint position is close to the final desired one, then there will be no time to perform the constant velocity phase, and only a two-phase acceleration-deceleration behaviour will be observed.

[1] Visit our webpage at www.ouroboros.org/thesis for additional information, including videos of the results obtained, related papers, and source code for the implementation of our results in your own Aibo robot.

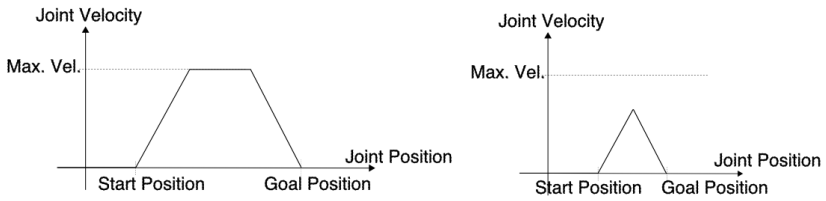


Figure 6.10: Figures showing the behaviour of a joint in the Webots simulator. In the left figure, the starting position of the joint is far away from the desired final position, and the joint therefore has to start from zero velocity, accelerate at maximum acceleration, maintain maximal velocity and then decelerate. In the figure on the right, the starting position is close to the desired final one, so the movement of the joint has no time to reach maximum velocity state, and just the two phases of accelerating-decelerating can be observed.

The problem with this approach is that when a position-controlled behaviour is implemented, all the inertia and velocity of the joint at the desired position is lost since it will reach that position at zero velocity. Even if this behaviour can prove useful for some determined control types, this is not in the case of walking, since a dynamic behaviour is required (inertia is used by the walking mechanism). Therefore, a control by velocity will be used. In a control by velocity, the controller does not indicate the position that the robot has to reach but rather its velocity. So in our experiment, the neural controllers will set the joint speed at each time step. In order to keep it simple only two velocitites will be allowed: maximum velocity in clockwise direction, and maximum velocity in counterclockwise direction. Even with this simple reduction of options, joints will have to handle acceleration and deceleration stages when changing the rotation direction.

6.4.2 Technical issues: neuron model

Technical analysis will begin with the type of ANN to be used in the IHU models for the generation of the gait controller. In previous examples, feedforward neural networks with or without hidden units were used. In this case a neural network able to capture dynamics is required due to the dynamic nature of the task to evolve. Requirements are:

- Since the task is a dynamical system in cycle-limit, the ANN has to provide internal states and/or continual dynamics.
- Mathematical complexity should be kept as low as possible in order to reduce the computational load.
- Oscillatory patterns should be generated from a tonic signal.

In [Reeve, 1999, Reeve and Hallam, 2005] there is an exhaustive analysis of the characteristics, advantages and drawbacks of different types of neural networks

for walking behaviours. Based on the listed requirements and the available options, the Continuous Time Recurrent Neural Nets (CTRNN) [Beer, 1995] model was selected. This is a model complex enough for the generation of acceptable walking gaits, and without the computational complexity of more accurated models. This type of network has also proven useful in generating walking behaviours using neural nets [Gallagher et al., 1996]. A CTRNN will be used in each IHU processing element to make it simple and homogeneous, even if in principle this would only be required for the actuators.

CTRNN's [Hopfield, 1984] are composed of a set of interconnected hidden neurons modelled as *leaky integrators* that compute the average firing frequency of the neuron (see figure 6.11) with output neurons following the standard perceptron output. CTRNN hidden unit outputs are computed using the following equations:

$$\begin{aligned} \tau_i \frac{dm_i}{dt} &= -m_i + \sum w_{ij} x_j \\ x_i &= \left(1 + e^{(m_i + \theta_i)}\right)^{-1} \end{aligned}$$

where m_i represents the mean membrane potential of the i-th neuron, that is, the output of the network; x_i is the short-term average firing frequency of the i-th neuron; θ_i is the neuron bias; τ_i is a time constant associated with the passive properties of the neuron's membrane; and w_{ij} is the connection weight from j-th neuron to i-th neuron. Calculation of each neuron output is performed using the Euler method for solving differential equations [Salzmann, 2003] with a given step of 96 ms. This value was found to be a useful tradeoff between the minimum value required to capture the dynamics and the time delay introduced by the execution of the software.

A neural net similar to the one shown in figure 6.11 was used for each IHU neural element; where the number of inputs equals the number of devices to control, i.e. 24; the number of hidden units is five, and the number of output units is one. Inputs represent the connections of a given IHU to the rest of modules, and the output is the answer from that IHU to its associated element (sensor or actuator).

For each of the networks' hidden units it is necessary to evolve: the connection weights for inputs and outputs, its bias term θ_i and its time constant τ_i. So, the genotype for each hidden unit will contain those values in a direct encoding scheme (see figure 6.12). Initially, the values of the parameters for all the neurons are randomnly created around certain values. Weights are initialised in the range from -16 to 16, as specified by the ESP algorithm. For the bias, the initial values of between -16 and 16 were also taken. For the time constant, values were between 0.5 and 10. Those values were decided based on experiments depicted in [Seys and Beer, 2004].

6.4.3 Single stage evolution of a walking gait

Before applying the progressive design method we will show how a direct evolution of the walking behaviour in one single stage leads to failure, with no walking

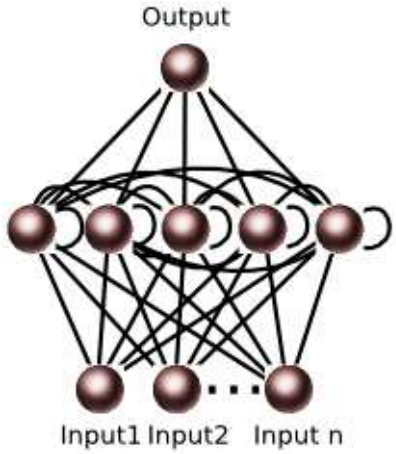


Figure 6.11: Schematics of the CTRNN used in the walking controller.

Figure 6.12: Genotype for the evolution of the leaky integrator inside the CTRNN number 1, with k inner connections. The Genotype encodes all the network parameters for the evolutionary process.

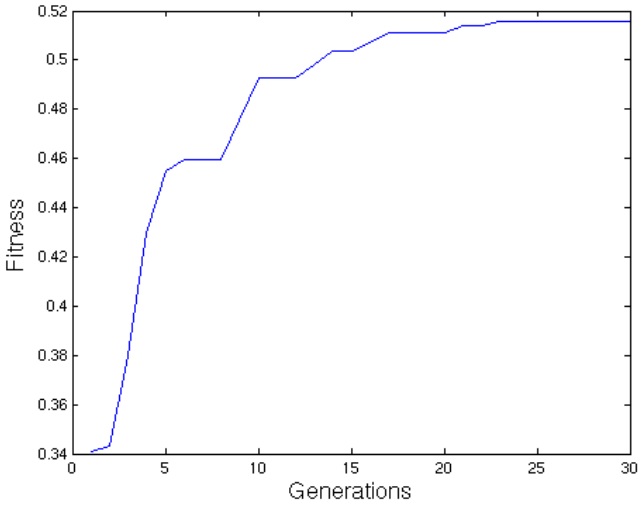


Figure 6.13: Mean fitness obtained for the single stage DAIR architecture walking behaviour.

behaviour emergence. Results shown in [Reeve, 1999] will be used as point of departure. There, the evolution of the walking behaviour of a simple simulated quadruped robot with 8 DOFs was achieved using the distance travelled by the robot as a fitness function.

6.4.3.1 A single stage using tactical modularity

Based on the results in [Reeve, 1999] for a more simple robot, the DAIR tactical modular architecture was applied to the walking task. A DAIR controller composed of 24 IHU's, that is, 24 CTRNN's, was designed. Hence, the search space to be faced for the algorithm consists of 3840 values to evolve: (24 inputs + 1 output + 5 recurrent connections + 1 bias + 1 time) * 5 hidden units * 24 nets. The evolution of the DAIR controller was tried for ten different runs, every run lasting for 30 generations. The fitness function used was,

$$fitness = distance\,advanced \tag{6.2}$$

None of the 10 controllers evolved was able to perform steps however, and all the runs ended in local minima with the robot lying on the ground trying to drag itself over the ground. In fact, this reptilian behaviour (see figure 6.14) allowed the robot to obtain a few points from the fitness function.

Using symmetry

As a second attempt, symmetry was applied in the same way as in [Reeve, 1999], since the robot morphology is symmetric. This means that only one half of the

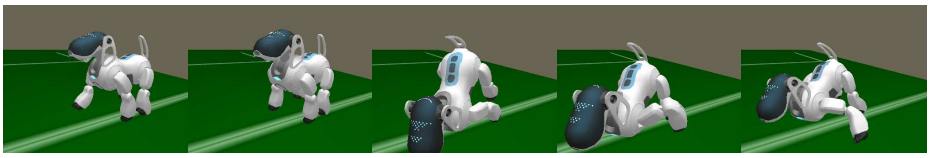


Figure 6.14: Sequence of the Aibo walking pattern obtained when a single stage evolution is used.

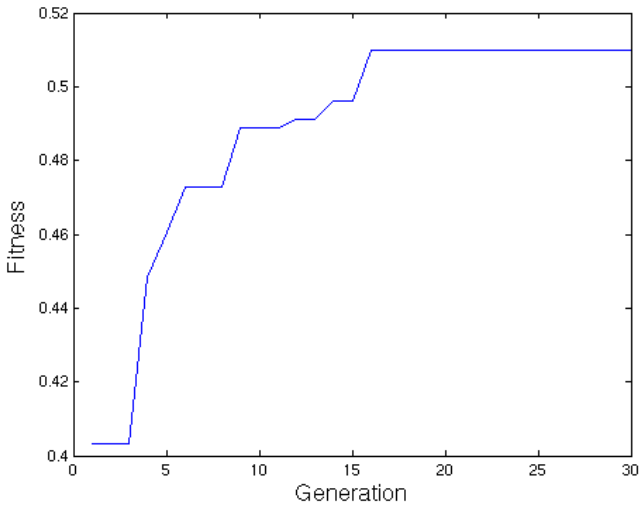


Figure 6.15: Mean fitness obtained for the single stage DAIR architecture walking behaviour, when using symmetry.

robot can be evolved, and, at the moment of testing, it duplicates the half controller to the other half of the robot. Using this plan, a reduction of half the search space is obtained. The same fitness function as in equation 6.2 was used. After repeating the 10 runs of 30 generations each, none of them was able to evolve the walking behaviour. Behaviours such as those in figure 6.14 were observed.

Using different fitness functions

Different fitness functions were used as an alternative. These were more complicated fitness functions that rewarded crossing legs and penalized jumping forward. Other more complicated add-on's to the formula were also divided in an attempt to generate an oscillation pattern, none of which was successful. An example of an additional formula used is presented:

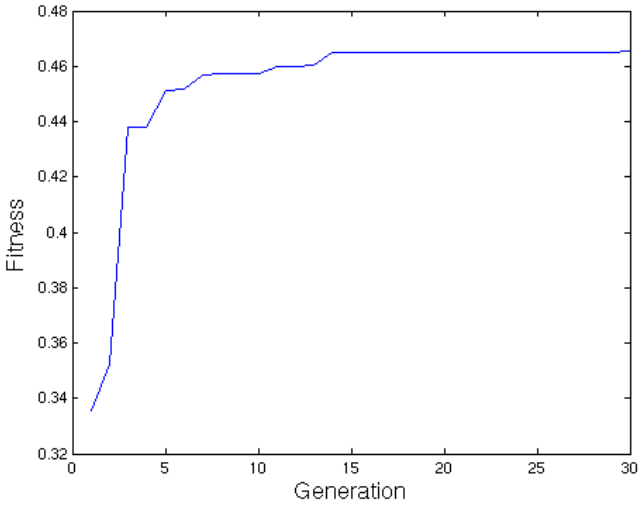


Figure 6.16: Mean fitness obtained for the single stage DAIR architecture walking behaviour with fitness equation 6.3.

$$fitness = \begin{cases} 0 & if\ -0.698 < J1\ joints\ < 0.261 \\ 0 & if\ 0.349 < J3\ joints\ < 1.658 \\ 0 & if\ less\ than\ three\ paws\ down \\ 0 & if\ paw\ not\ down\ after\ up \\ distance * height & otherwise \end{cases} \quad (6.3)$$

Results

Even though an evolved walking behaviour for a certain simulated robot was reported in [Reeve, 1999], we were not able to evolve it for the simulated Aibo. Furthermore, we know of no other report indicating that it has been achieved with an Aibo robot using only the distance travelled as a fitness function. The main differences between our case and that of [Reeve, 1999] may be that of Aibo's simulation; also there is an additional degree of freedom on each of its legs, moreover, Aibo has a very different physical structure which includes a heavy chest, a head and a neck, as well as having feet with a shape that do not only touch the ground at a single point, as in the case of [Reeve, 1999]. Furthermore, in the case of [Reeve, 1999] no sensor information was taken into account. All these differences may explain the differences between our results and those of [Reeve, 1999]. No more research on this subject was performed.

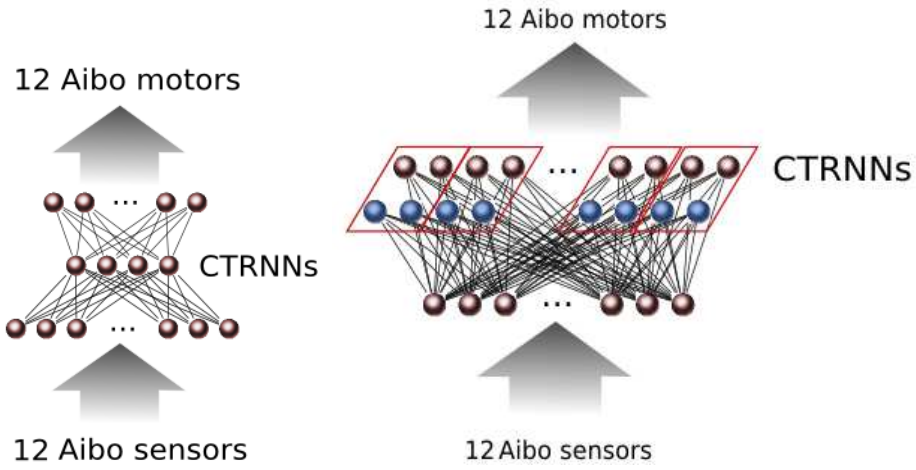


Figure 6.17: Left: monolithic architecture for the evolution of the Aibo walking behaviour in one single stage. Right: the emergent modular architecture used for the evolution of the same behaviour.

6.4.3.2 A single stage evolution using other architectures

Before attempting the application of the progressive design method, we tried to use the other successful architectures from chapter 5: the monolithic architecture and the emergent modular (see figure 6.17). In the first case, the neural controller had 12 inputs coming from the joint sensors, and 12 outputs going to the joint motors. Unfortunately, the algorithm was clueless on this task for a fitness function that rewarded controllers able to go forward, without falling down while generating oscillations in the legs (that behaviour is described in equation 6.3). However, none of the controllers achieved the generation of a single step. The same unsuccessful result was obtained for the emergent modular architecture. In this case the neural controller had 12 inputs and 24 outputs. The same 6.3 equation was used as the fitness function.

Results obtained suggested more than a general problem of search space dimension, but rather a problem of bootstrap, that is, the task to evolve is so complex that processes are not able to find an evolutionary path from a completely random controller for the full walking controller (if the path ever exists!). It seems unlikely that any amount of knowledge will make any of the architectures evolve the desired behaviour if this knowledge is directly applied. Hence, we conclude that it is then necessary to use a progressive design for the evolution of the walking gait.

6.4.4 The CPG concept for walking robots

One of the most important features of the progressive design method is the use of external knowledge that help decide the evolutionary path to implement,

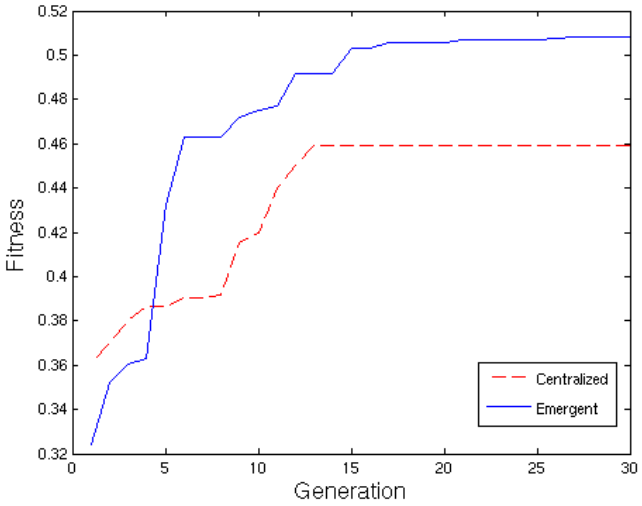


Figure 6.18: Fitness obtained by the single stage monolithic architecture walking behaviour and a single stage emergent modular architecture walking behaviour.

that is, the number of stages and their content; in evolutionary terms. For this purpose, knowledge on walking robots is required. The concept of Central Pattern Generator (CPG), the main concept used for the generation of a walking pattern, is described here.

It has been extensively shown [Grillner, 1985, Collins and Stewart, 1993, Calancie et al., 1994] that animals perform rhythmic movements such as walking or running by means of groups of neurons, known as CPG's, which generate an oscillatory pattern from a tonic input signal. Depending on the tonic signal value, CPG's can change its frequency of oscillations as well as its amplitude. It is presumed that humans could also be controlled by such systems. Central pattern generators is a methodology that has been widely used for walking robots [Lewis et al., 1992, Ijspeert, 1998, Kimura et al., 1999, Billard and Ijspeert, 2000, Ijspeert, 2001, Kimura et al., 2001, Ijspeert, 2002, Mojon, 2004, Markelic, 2005, Manoonpong et al., 2005, Manoonpong et al., 2007].

CPG's can be artificially created using ANN's. These CPG's will be the ones to be implemented in the robot controller by using the distributed architecture. In this work, a CPG will be implemented for the control of each joint. This means that *IHU modules will implement the CPG function.* Aibo's joints are composed of a sensor which obtains the position of the joint at each instant, and an actuator, which moves the joint. Since each joint has an associated sensor and motor, the IHU's of the sensor and motor will work together to generate the oscillation that would drive the robot joint. Hence, a CPG is implemented for each joint by coupling an IHU for the joint sensor, and an IHU for the joint actuator (see figure 6.20).

Differences exist however between our architecture implementation and real CPG control in animals. Firstly, in the case of real CPG's, only contiguous CPG's are connected to each other [Grillner, 1985]. In the DAIR implementation however, all the IHU's are interconnected , which in practical terms means that all CPG's are connected to all CPG's. Secondly, it is not clear whether real CPG's have direct connections to sensors or not. There is indeed a reflex system which uses sensor information to modify CPG behaviour, but how this is achieved still remains largely unknown [Ijspeert, 2002]. In fact, in artificial CPG-based walking systems the reflex mechanism is usually implemented as an additional external system that works beside the oscillatory system [Klavins et al., 2001].

In order to reduce the search space for the evolutionary algorithm not all the connections of the DAIR controller are evolved at the same time as stated in section 6.4.3. A progressive design will be performed, which guides the evolutionary process towards a correct solution. The different stages for the generation of the walking gait will be: generation of CPG oscillators, where a segmental oscillator is evolved for each type of joint; generation of two coupled CPG's in counterphase; generation of a layer of four CPG's; and coupling of three groups of four CPG's to obtain the final walking behaviour.

6.4.5 Progressive design of a walking gait

The evolutionary strategy for the generation of the controller intensively uses the CPG concept: the robot controller will generate a group of CPG's, - one per each joint; each one driving its corresponding joint with an oscillation signal. Joint oscillators will be coordinated in the appropriate way so as to produce a walking gait.

A fundamental point is that the generation of the oscillatory patterns will not be performed externally of the robot, as has been the case in other works. That is, the evolution of the oscillatory pattern is performed in the (simulated) robot itself. This allows the neural nets to capture the dynamics of the (simulated) robot, producing an oscillatory signal that takes into account inertias, accelerations, etc. These are all important features when dealing with a robot as large and as complex as Aibo.

6.4.5.1 Progressive design strategy

The key point for the evolutionary strategy is that of generating an oscillation for each joint, each one with its own required phase, and then to couple them all into a single controller. Once this point became clear, its actual implementation was divided into four stages: in the first stage, a controller that performs an oscillation is generated for each joint using its associated motor and sensor. At the end of that stage each joint has its own an independent oscillator that drives it. In the second stage two oscillators from the same joint type but from different legs are coupled in such a manner that they generate a single controller driving both joints with an oscillation, but with a required phase

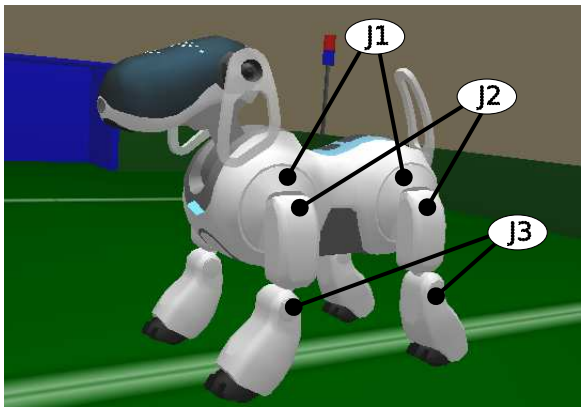


Figure 6.19: Details of the three different joint types for the robot legs.

shift between them. In the third stage, two groups of two coupled oscillators are again coupled, obtaining a group of four oscillatory joints driven by one single controller. These three stages are repeated for the three types of joints that the robot has (J1, J2 and J3, see figure 6.19), which leads to a situation where three separated and independent controllers exist, with each one driving four joints in a synchronized oscillation. The fourth stage evolves the connections between those three controllers, generating a single controller for the whole robot, which is then finally able to walk.

6.4.5.2 First stage: generation of the joint oscillator

The aim of this stage is to obtain a controller for a joint by generating an oscillatory pattern for each type of robot joint. Joints in the robot's legs are of three different types, which we will call J1, J2 and J3. J1 is in charge of the rotatory movement of the shoulder, J2 of the lateral movement of the shoulder and J3 of the knee movement. Physical joints are controlled by using different PID controllers, and, in addition, their movement limits are different. For this reason, a different type of oscillator must be implemented for each joint type so that the three types of oscillators will be evolved in separated evolutionary processes in this stage. Nevertheless, the process for the generation of each type is exactly the same, with the only practical difference being the range of movements and the way the leg will be moved. The sensor and actuator vectors for this stage *t1* are then as follows:

$$\mathbf{y}_{t1}^{J1} = \{y_{S-J1LF}\} \quad \mathbf{y}_{t1}^{J2} = \{y_{S-J2LF}\} \quad \mathbf{y}_{t1}^{J3} = \{y_{S-J3LF}\}$$

$$\mathbf{u}_{t1}^{J1} = \{u_{M-J1LF}\} \quad \mathbf{u}_{t1}^{J2} = \{u_{M-J2LF}\} \quad \mathbf{u}_{t1}^{J3} = \{u_{M-J3LF}\}$$

An oscillator is implemented for each joint by coupling two CTRNN networks: one for the joint sensor and another for the joint actuator (the motor).

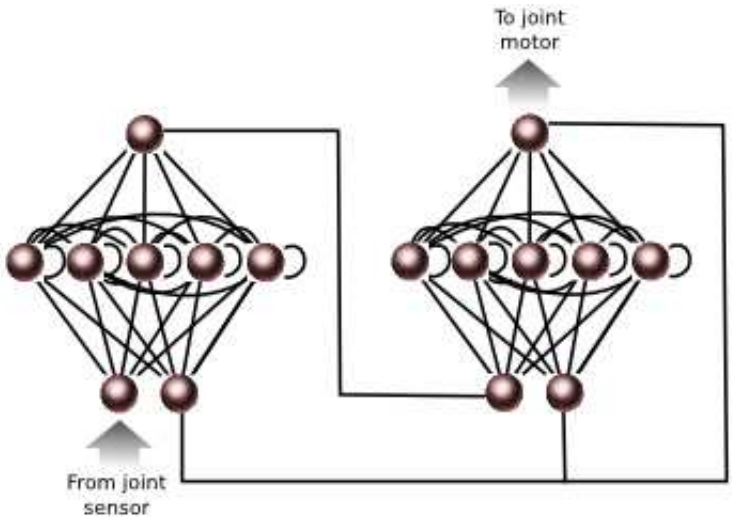


Figure 6.20: Schematics of the coupling between two neural nets for the control of a joint in stage *t1*. The figure shows the coupling between the joint sensor IHU and the joint motor IHU.

The DAIR architecture described in chapter 4 is applied to a single joint composed of two devices. The resulting controller is shown in figure 6.20. Both nets are interconnected as specified by the architecture, but each one is in charge of a different element: the sensor net is in charge of the sensor, and the motor net in charge of the motor. The sensor reading is taken and entered into the IHU sensor at each time step of the evaluation process. Then a certain output is processed and sent to the IHU actuator. The output of the IHU actuator specifies the motor's velocity to be applied, once de-normalized, it is applied directly to the motor.

The weights of the nets are evolved using both the ESP algorithm and a fitness function rewarding the production of an oscillatory pattern in the motor joint. The genotype used is depicted in figure 6.22. The exact pattern to be obtained is not specified, but it is periodic and between certain oscillatory limits. The following fitness function was defined for the evolution of such oscillations:

$$mean = \frac{1}{N}\sum x_i$$

$$fitness = \frac{1}{N}\sum (x_i - mean)^2 \qquad (6.4)$$

with x_i being the joint position at time step i. Basically, fitness function calculates the variance of the trajectory followed by the joint. Thus, the evolutionary process will try to maximize it, and the value of the variance will be maximum when the position of the joint changes from one limit to the other.

Table 6.1: Limits used for Aibo joints oscillations. It includes a calculated mean value of the range for each joint. This mean value establishes the central position of the joint around which the joint will have to oscillate. Values are in radians.

Joint	Max	Min	Mean
J1, fore	0.3936	-0.5837	-0.0950
J2, fore	0.3702	-0.2163	0.0769
J3, fore	1.1732	0.1435	0.6583
J1, rear	0.0059	-0.7848	-0.3894
J2, rear	0.4215	-0.2163	0.1026
J3, rear	1.6599	0.9907	1.3253

A single oscillation was obtained within the full range of the joint by using the previous fitness function, that is, the joint oscillated one single time from one limit to the other. Aibo joints can oscillate between very large limits, but these are too large for an appropriate walking behaviour, and they must be limited by defining a number of oscillation bounds. In order to keep it simple, limits were defined by observing the walking bounds for the Sony's default walking style. These limits are included in table 6.1.

A new fitness function was defined to reward regular oscillations within the joint bounds. The system is now required to generate a joint movement around the mean value, with maximal variance within the limits for each joint. The fitness function is then a product of two factors:

$$fitness = fit\_var * fit\_cross \tag{6.5}$$

where fit_var is the variance of the joint position during the 200 evaluation steps, as calculated in the 6.4 equation, and fit_cross is the number of times that the joint crosses in its movement through the mean positional value. The addition of the second term to the fitness function favours the final oscillation obtained not being limited to a single oscillation. This term also propriciates the generation of very high frequency oscillations. However, with limited time and speed, variance will decrease when frequency increases, and so the first term regulates the increase of the second one. In any case, in order to avoid sub-optimals with too high frequency oscillations, a limitation in the number of possible crossings was established at 20.

In addition, each fitness factor was limited by a function that linearly varies between 0.01 and 1.0 when its corresponding variable fluctuates between a good and a bad boundary (see table 6.2). In other words, normalization was applied to each factor in order to obtain a fitness value between 0.01 and 1 for either bad or good results. This transformation allowed a clear vision of the evolutionary state at any time. The transformation is calculated using the following formula:

$$F(f) = 0.99 \cdot \frac{f - G}{G - B} + 1$$

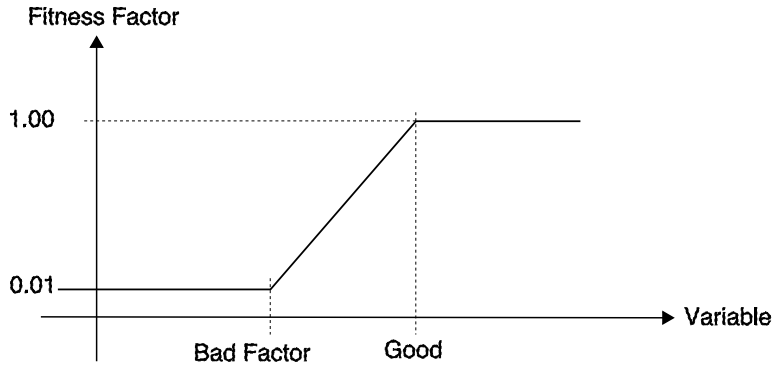


Figure 6.21: Transformation function for fitness normalization. *Bad* and *good* values depend on the fitness function, and it is provided below in different tables.

Table 6.2: Variables and boundaries for the first stage's fitness function.

Function	Variable	Boundary [bad,good]
fit_var	variance of the joint	[-1.0 , 0.47]
fit_cross	n. of crossings	[0.0 , 20.0]

where B and G are for the *bad* and *good* boundaries. Boundary values for each stage are provided in the tables below.

Results

Seven runs were carried out for each type of joint starting with different initial random populations. Each run was composed of 200 simulation steps of 96 ms. After 8 generations all runs converged to networks maintaining oscillations within the range specified.

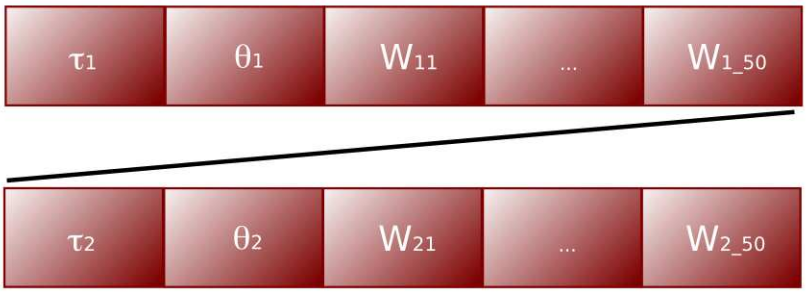


Figure 6.22: Genotype for the first stage of the Aibo walking.

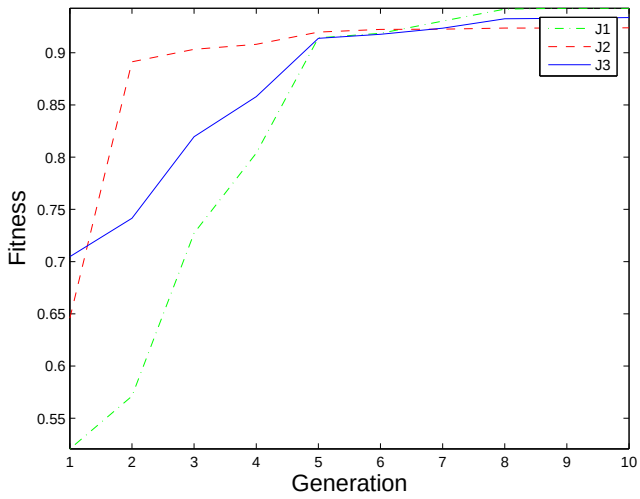


Figure 6.23: Fitness evolution of the three types of Aibo joints in one single joint (first stage), averaged over seven evolutionary runs.

6.4.5.3 Second stage: generation of two coupled oscillators

In the previous stage, three different and independent controllers were obtained, each one for the control of one type of the robot's joints. Each controller was composed of two tactical modules: one for the control of a sensor and another for the control of a motor. The aim of the second stage is to obtain a controller for two joints of the same type, which will result in a coupled oscillation within a given phase relationship.

In this case, the controller will now be composed of four neural networks (for each type of joint): two controlling the two motors and two controlling the two sensors. Since we evolved the controller for one joint and its sensors in the previous stage. First solution would be to evolve the neural modules for the newly added motor and sensor, as well as the connections between these newly added modules and those already existing for the previous joint, in the same way as depicted in figure 6.1.

A quicker and simpler solution exists however, due to the fact that the new joint to control is of the same type as the one controlled in the first stage. It consists of performing a duplication of the modules created in the previous stage for the new joint and just evolving the *connections between modules*. The oscillator controlling one type of joint from the previous stage will be copied to control the joint of the same type, i.e. the joint that is opposite the original one, thereby obtaining two isolated oscillators; each one like the one displayed in figure 6.20. Connections between both modules are then established in order to apply the architecture definition, and only these connections need to be

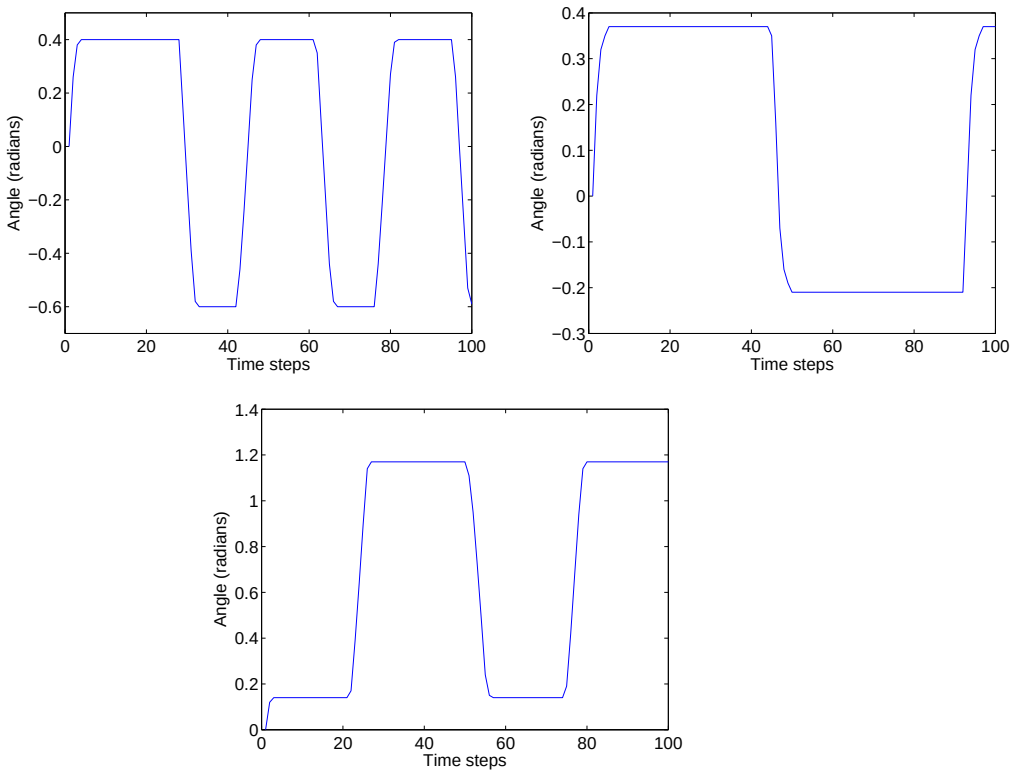


Figure 6.24: Oscillatory patterns obtained for all three types of joints in the first stage. Each joint oscillates in its own range.

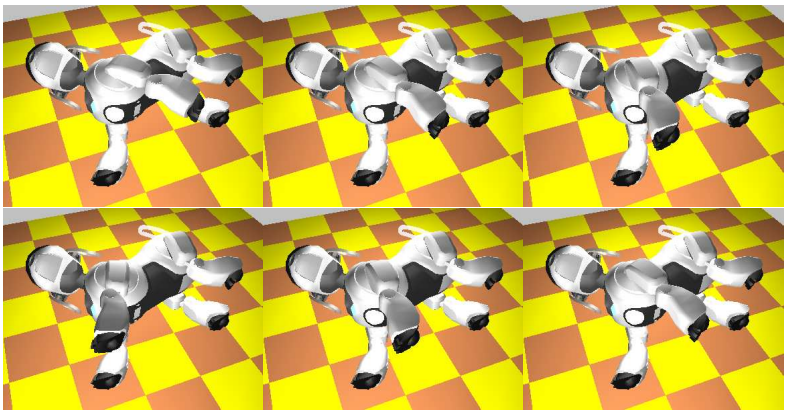


Figure 6.25: Sequence of the oscillatory movement obtained in simulation for the J1-type joint (left fore leg in the figure). During the first stage the robot is lying on his side to allow for the free movement of the joint. The evolution of the other joint types was performed with the same setup.

Table 6.3: Phase relationship, according to [Collins and Richmond, 1994], for three common quadrupedal gaits.

Leg	Walk	Trot	Bound
Left-Fore	0^{o}	0^{o}	0^{o}
Right-Fore	180^{o}	180^{o}	0^{o}
Left-Rear	270^{o}	180^{o}	180^{o}
Right-Rear	90^{o}	0^{o}	180^{o}

evolved in this stage (neither the internal connections of the neurons nor the time constant and bias obtained from the previous stage will be evolved). A phase relation of π rad between those two legs (in all types of joints) will be required, as stated in table 6.3. It is important to note that the three types of oscillators are completely independent yet from each type of joint at this stage, and evolved in different evolutionary processes. So, each evolutionary process therefore has to cope with a reduced search space only related to its associated joint type. Hence the vectors are:

$$\mathbf{y}_{t2}^{J1} = \{y_{S-J1LF}, y_{S-J1RF}\}\ ,\ \mathbf{u}_{t2}^{J1} = \{u_{M-J1LF}, u_{M-J1RF}\}$$

$$\mathbf{y}_{t2}^{J2} = \{y_{S-J2LF}, y_{S-J2RF}\}\ ,\ \mathbf{u}_{t2}^{J2} = \{u_{M-J2LF}, u_{M-J2RF}\}$$

$$\mathbf{y}_{t2}^{J3} = \{y_{S-J3LF}, y_{S-J3RF}\}\ ,\ \mathbf{u}_{t2}^{J3} = \{u_{M-J3LF}, u_{M-J3RF}\}$$

The fitness function will reward phase differences between the legs close to the π rad, as well as a continuous oscillatory movement of both legs. Hence, a fitness function containing these three components is proposed: the first two components correspond to the fitness function that measures the oscillation of

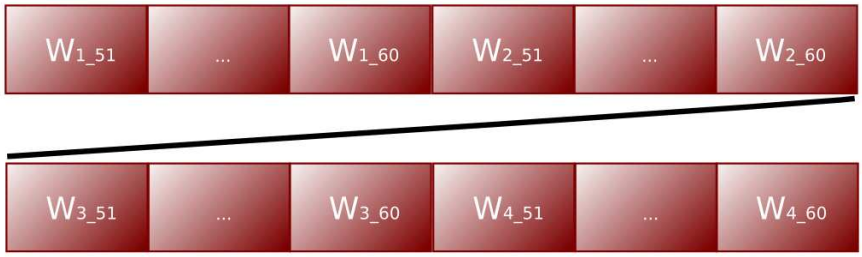


Figure 6.26: Genotype for the second stage of the Aibo walking

Table 6.4: Variables and boundaries for the fitness function in the second stage

Function	Variable	Bound [bad,good]
fit_cross1	crossings joint 1	[0.0 , 20.0]
fit_cross2	crossings joint 2	[0.0 , 20.0]
fit_var	variance of the 2 joints	[-1.0 , 0.47]

each joint independently, and the third component corresponds to the fitness factor that measures the variance between the movements of both legs, and tries to maximise it.

$$fitness = fit\_cross_1 * fit\_cross_2 * fit\_var \tag{6.6}$$

where fit_var is the variance for the difference of positions between both legs during the 400 evaluation steps, and fit_cross_1 and fit_cross_2 are the number of crossings that each joint has performed through its mean position value. Similar to the fitness function for the first stage, these factors vary between 0.01 and 1.0 when their corresponding variable fluctuates between a good and a bad boundary (see table 6.4). The genotype evolved is depicted in figure 6.26.

Results

Seven runs were carried out for each type of joint to evolve the connections between CPG's. Each run was composed of 400 simulation steps of 96 ms. After 9 generations 90% of the networks performed a counter-phase oscillatory pattern.

6.4.5.4 Third stage: coupling the oscillation of four joints of the same type

In this stage, the control for the rear two joints of the same type is added to the controller. This means that four new neural modules will be added to the modular controller: two for the control of the two rear joints, and two for the two rear sensors. However, since the two front joints have the same phase

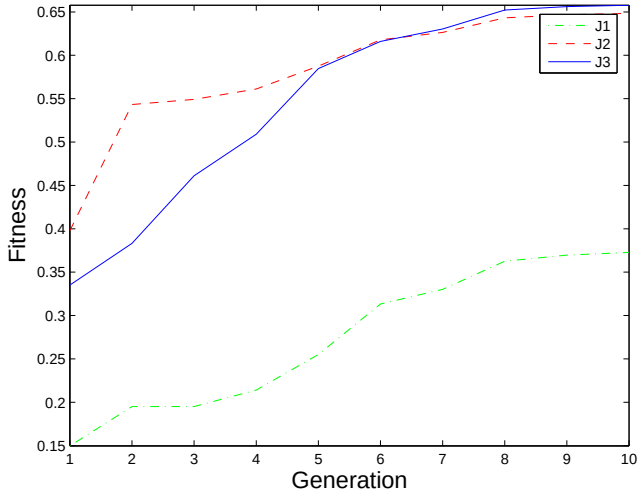


Figure 6.27: Evolution of the fitness for the three types of joints in the second stage, averaged over seven evolutionary runs.

relationship for a walking gait as the two rear joints have, it is possible to clone the controller obtained in the second stage for the front joints to control the rear joints, and then evolve only the connections between them. Hence the vectors are:

$$\mathbf{y}_{t3}^{J1} = \{y_{S-J1LF}, y_{S-J1RF}, y_{S-J1LR}, y_{S-J1RR}\}$$
$$\mathbf{y}_{t3}^{J2} = \{y_{S-J2LF}, y_{S-J2RF}, y_{S-J2LR}, y_{S-J2RR}\}$$
$$\mathbf{y}_{t3}^{J3} = \{y_{S-J3LF}, y_{S-J3RF} y_{S-J3LR}, y_{S-J3RR}\}$$
$$\mathbf{u}_{t3}^{J1} = \{u_{M-J1LF}, u_{M-J1RF}, u_{M-J1LR}, u_{M-J1RR}\}$$
$$\mathbf{u}_{t3}^{J2} = \{u_{M-J2LF}, u_{M-J2RF}, u_{M-J2LR}, u_{M-J2RR}\}$$
$$\mathbf{u}_{t3}^{J3} = \{u_{M-J3LF}, u_{M-J3RF}, u_{M-J3LR}, u_{M-J3RR}\}$$

Since four new modules are added for the control of the two rear joints, it is necessary to evolve 4 connections per network with a total number of 8 IHU's per joint type. The evolution of the new connections is performed using a fitness function which rewards controllers that achieve an oscillation of all the legs with a given phase relationship between them, as specified in the 6.3 table. For a typical walking gait the relation from left to right and from front to rear is 0^{o},180^{o},270^{o},90^{o}. In the previous stage, due to the fact that the phase relationship was of 180^{o} a simple algorithm for the calculation of that relation was performed; based on the calculation of the variance between both oscillations. At this stage, more complex phase relationships are required and the same trick cannot be implemented. Hence, for the calculation of the oscillation phase of each joint we implemented a DFT algorithm which calculated the

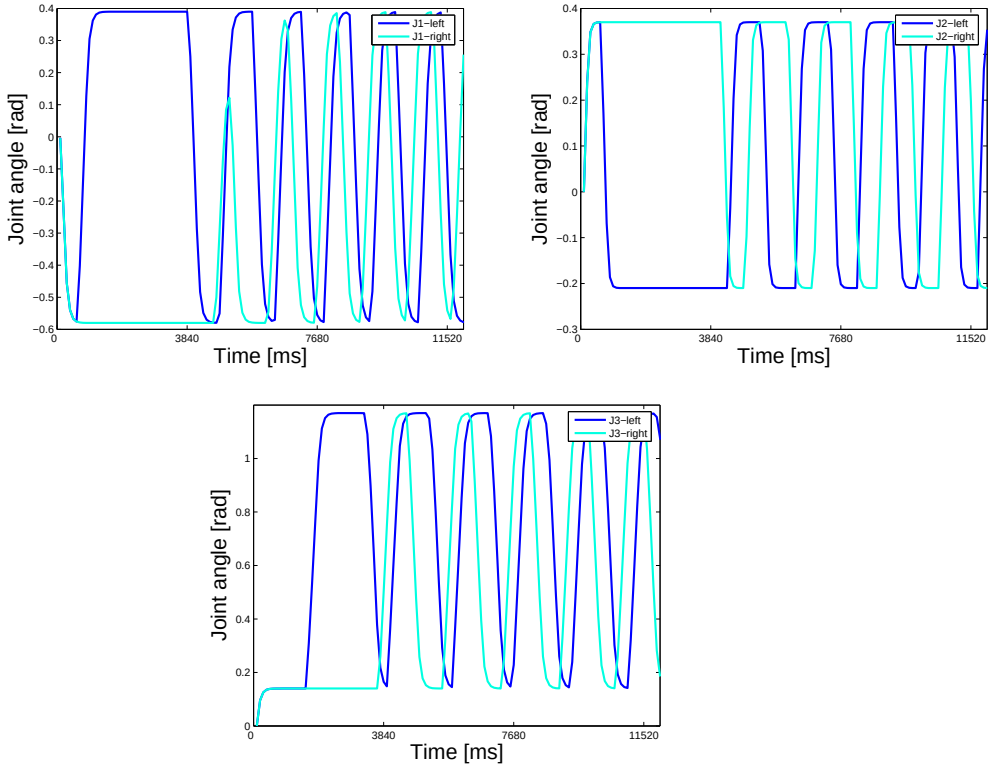


Figure 6.28: Oscillations obtained for each type of joint when two joints are coupled in second stage. From left to right oscillations for joint types J1, J2 and J3.

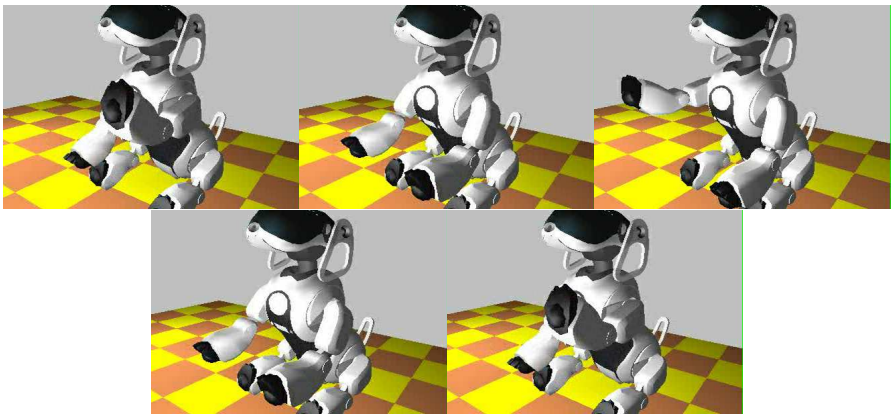


Figure 6.29: Sequence of the oscillation obtained in simulation for the J1 joint type during second stage. The robot is sat on its backside to allow for a free movement of the joint. The evolution of the other joint types was performed with the same setup and similar behaviours were observed, as indicated in figure 6.28.

main harmonic phase of the oscillation signal produced by each joint. The main harmonic phase of each joint oscillation was used for both comparison and to assess the oscillation relationship.

Hence, making use of that algorithm, the fitness function designed to obtain coordination was composed of three parts: a part for each leg that expresses the oscillation requirement; a part that expresses the maximal variance requirement between the fore legs; and a final part that expresses the maximal variance requirement between fore/rear differences. This has been specified in the following fitness function:

$$fitness = fit\_cross * fit\_oscil * fit\_phases \tag{6.7}$$

where fit_cross is the product of the number of crossings for each joint performed through their mean positional value, fit_oscil is the variance of the left fore joint which indicates how well that joint oscillates, assuming that if this joint oscillates the others must follow, and fit_phases is the part of the fitness that indicates the phase relationship between all the joints. Similar to the fitness function for the previous stage, these factors vary between 0.01 and 1.0 when their corresponding variable oscillates between a good and a bad boundary (see table 6.5). The genotype evolved is depicted in figure 6.30. The Aibo robot was placed upside-down for the evolution of the coupling between the four joints in order to allow for a free motion of the joints(figure 6.33).

Results

The evolutionary process was carried out seven times. Each evolutionary run was composed of 400 simulation steps of 96 ms each. After 14 generations, 92%

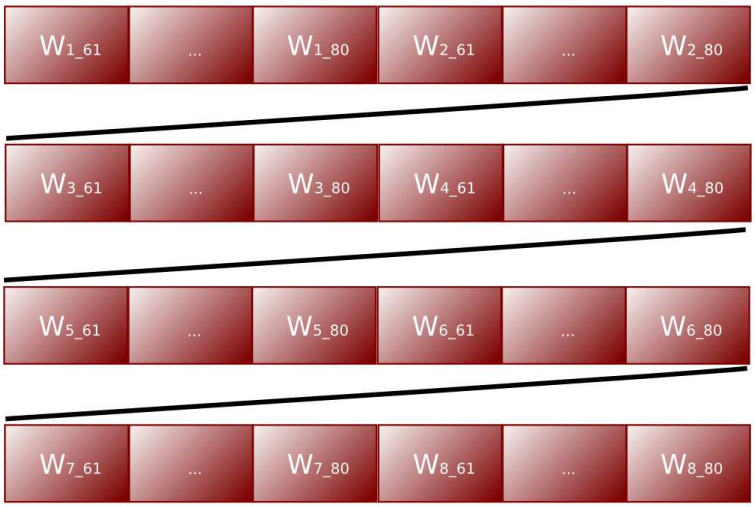


Figure 6.30: Genotype for the third stage of the Aibo walking.

Table 6.5: Variables and boundaries for the fitness function in the third stage.

Function	Variable	Bound [bad,good]
fit_cross	crossings product for joints	[0.0 , 27000.0]
fit_oscil	fore left joint variance	[-1.0 , 0.47]
fit_phases	joints phase relationship	[0.0 , 1.0]

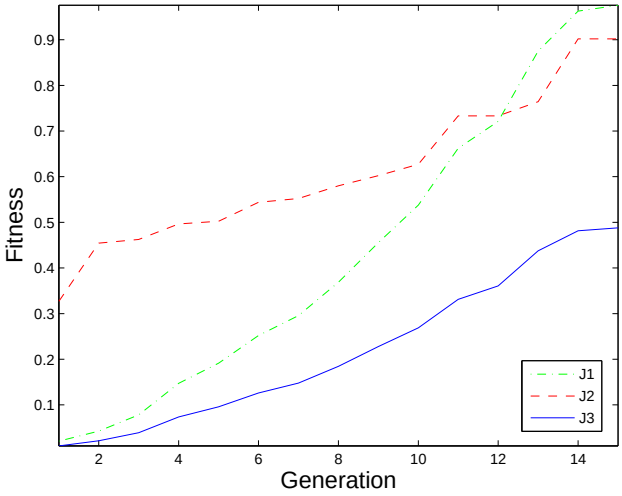


Figure 6.31: Evolution of the fitness for the third stage, through generations. Fitness is averaged over seven runs.

of the networks generated the typical oscillatory walking pattern 0^{o}, 180^{o}, 90^{o}, 270^{o} (for the leg sequence fore_left, fore_right, rear_left, rear_right).

Since each of Aibo's legs is composed of three different types of joints, the same process explained in sections 6.4.5.2 and 6.4.5.3 was performed for each joint type, resulting in three different controllers of coupled joints oscillating with the required phase relationship. At this point each controller for each type of join (J1, J2, J3) was independent from the other controllers in terms of their oscillation. The oscillatory patterns obtained for each type of joint can be seen in figure 6.32.

6.4.5.5 Fourth stage: coupling between types of joints

The last stage is the coupling between the three groups of neural controllers obtained. three different oscillating modular controllers were obtained from the previous stage; one per joint type, with four joints of the same type oscillating together in a *walking* phase relationship. What is now required is to connect the three layers in order to obtain a coordination between the different joint types to enable the robot to walk, and to complete the architecture as a whole. The next step is the evolution of the connections between the three groups of controllers. In terms of walking, connection between groups should produce coordination between the different types of joints that have been evolved separately. The vectors change now from three independent ones to a single one:

$$\mathbf{y}_{\text{tg}} = \{y_{S-JkLF}, y_{S-JkRF}, y_{S-JkLR}, y_{S-JkRR}\}_{k=1}^{3}$$

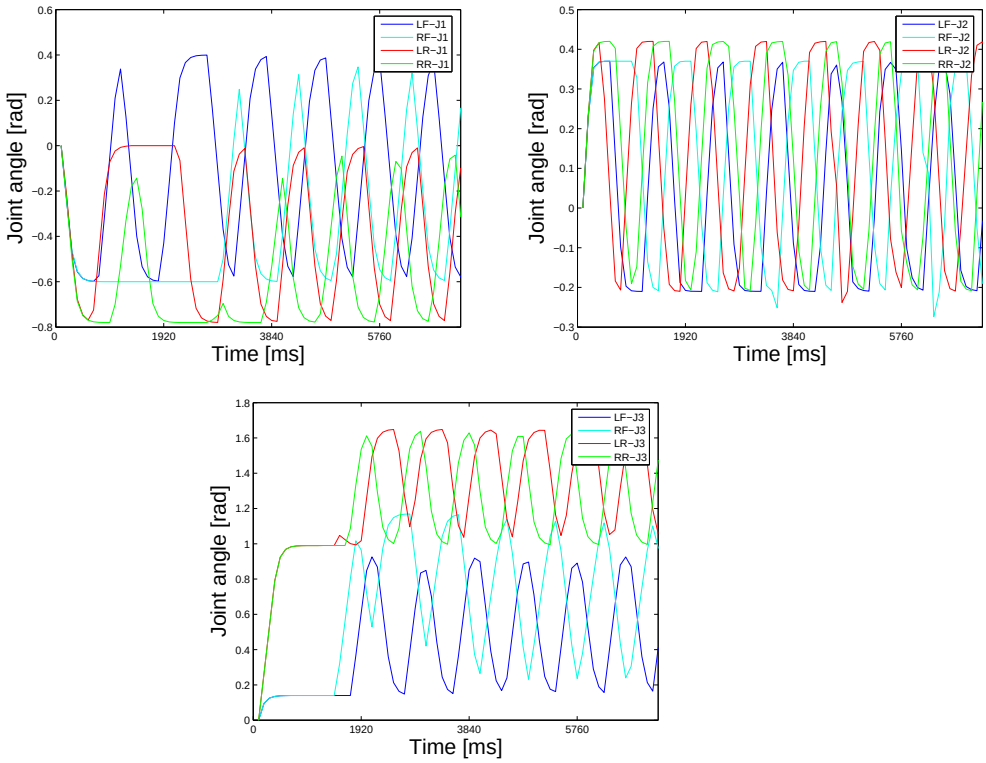


Figure 6.32: Oscillations obtained for all types of joints in all legs.

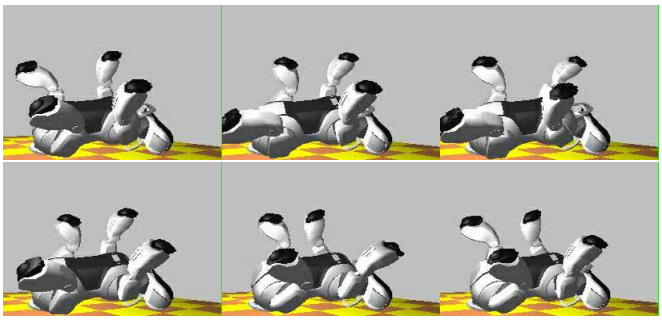


Figure 6.33: Sequence of the oscillation obtained in simulation for the J1 joint type during stage 3. The robot is lying on its back to allow for free movement of the joints. The evolution of the other joint types was performed with the same setup, and similar behaviours were observed, as indicated in figure 6.32.

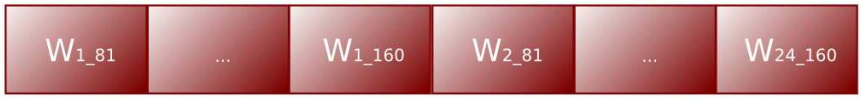


Figure 6.34: Genotype for the fourth stage of the Aibo walking.

$$\mathbf{u}_{\text{tg}} = \{u_{M-JkLF}, u_{M-JkRF}, u_{M-JkLR}, u_{M-JkRR}\}_{k=1}^{3}$$

Connections between the three groups of controllers imply that 16 new inputs will be added to each IHU neural module. These inputs represent the connection to the other 16 modules of the other two groups. Only these connections between groups are evolved to generate the required coordination between the groups for the generation of stable walking. As a first approach, the coordination between groups was attempted by evolving the connections using a simple fitness function consisting of the distance walked by the robot. However, the walking behaviour obtained by this approach, even if correct, was very sudden and induced instabilities that at times made the robot fall. Analysing the behaviour obtained we observed that coordination between the groups was correctly achieved, but that some of the joints had lost their oscillation pattern due to the new connections affecting the successfully evolved oscillatory controller.

For this reason, a new fitness function was proposed where the oscillation of the joints was still imposed, together with the distance walked. If the robot does not fall over the fitness function is composed of two multiplying factors: the distance d walked by the robot in a straight line and the phase relationship between the different joints. If the robot does fall over the fitness is zero.

$$fitness = \begin{cases} d * fit\_phases & when \quad final \quad height > 0 \\ 0 & otherwise \end{cases}$$

where the phase relationship factor is a normalized product of the phase relationship for each of the layers, each of which is calculated as explained in section 6.4.5.3.

Results

A walking behaviour was obtained after 30 generations for around 87% of the populations. The best walking sequence obtained is shown in figure 6.36.

Once this walking behaviour was obtained in the simulator, the resulting ANN based controller was then transferred to the real robot using the Webots simulator cross-compilation feature that we developed (see appendix A for a complete description). This cross-compilation process takes the exact controller developed in the simulator (the best of those evolved), and automatically translates it into Aibo's OPEN-R code, which is then executed in the real robot. The result was an Aibo robot that walks in the same manner as the simulated robot, with a few minor differences. The walking sequence obtained is shown in figure 6.37.

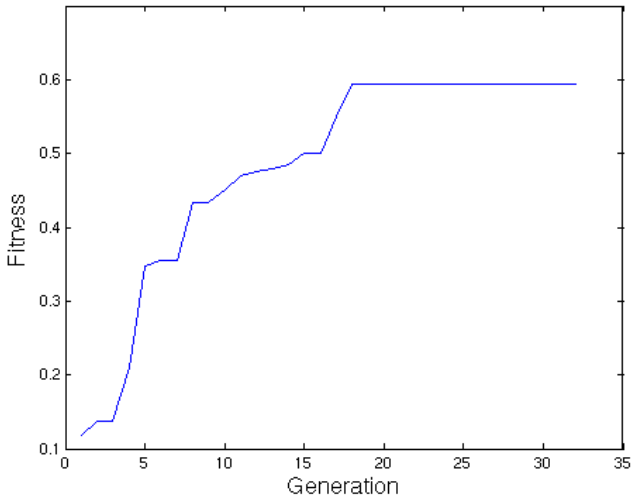


Figure 6.35: Fitness evolution of the fourth stage through generations; averaged over seven evolutionary runs.

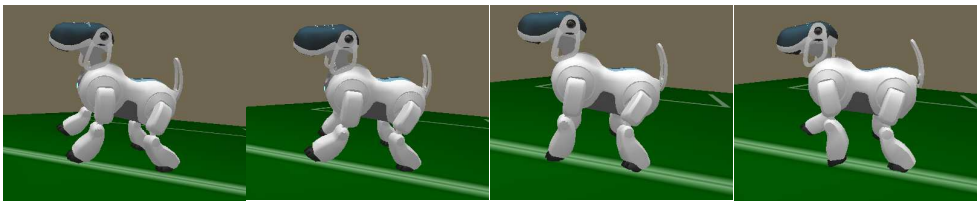


Figure 6.36: Simulated Aibo walking sequence.

Figure 6.37: Real Aibo walking sequence.

6.4.6 Remarks on the results obtained

The implementation of each CPG was carried out using two neural nets: one in charge of the sensor and one in charge of the actuator for every joint. Formally, the implementation of a CPG does not require the use of sensor inputs, but the introduction of the sensor networks could provide the system with a reflex system that could be helpful in unpredicted circumstances [Ijspeert, 2002]. The architecture introduced here integrates feedback from the sensor into the CPG; its benefits have not yet been studied and will form part of our future research. In particular, the reflex system of the DAIR architecture is integrated into the CPG walking structure itself, not being a separate system, and could benefit the walking style when dealing with uneven terrain or small obstacles, permitting the robot to adapt to them and carry on walking.

The gait obtained could most probably be optimised in terms of speed, energy consumption, and aesthetics by fine-tuning the fitness functions, including penalties for energy consumption and minimum inertia in the joints, amongst other factors.

A few minor differences were detected in the transfer from the simulator to the real robot. Whereas the simulator was able to generate a robot that walked in a straight line, the real robot showed a slight tendency to turn to the right due to the friction generated by the rear righthand leg. Several friction parameters were tested on the simulator in order to find the best friction coefficients relationships, but the real robot ended on a round trajectory in each case. This indicates that other factors are affecting the process of transference from simulator to real robot. This is a subject to be studied in the future.

6.5 Discussion

The progressive design method allows for the evolution of controllers for complex robots in complex tasks in complex robots. However, the process is not performed in a completely automatic way as aimed for by the evolutionary robotics approach. Instead, a gradual *shaping* of the controller is performed, with human training driving the learning process by presenting increasingly complex learning tasks, and deciding the best combination of modules over time, until the final complex goal is reached. This process of human shaping seems unavoidable if a complex robot body, sensors/actuators, environment and task are imposed beforehand. This point has also been suggested by other researchers [Urzelai et al., 1998, Muthuraman et al., 2003]. Progressive design, on the other hand, differs from other approaches by implementing modularity at the device level as well as the learning level, thereby allowing for better flexibility in terms of the shaping. The main reason for this, is that due to the modularization at the device level the designer can select at any evolutionary stage which small group of sensors and actuators will participate under which task, and then evolve only those modules. This would not be possible in a complex robot if modularization at the behavioral level is applied.

Progressive design can be seen as an implementation of the incremental evolution technique but with better control of who is learning what at each stage of the evolutionary process. If incremental evolution were used in a controller with several inputs and outputs which control every aspect of the robot, it may end up producing genetic linkage, and learning some behaviours in the early stages would prevent the learning of other behaviours in the next steps as the controller is so biased that it cannot modify itself to accommodate the new behaviour whilst keeping the old one working. This effect is especially important in complex robots where several motors need to be coordinated. Instead, the use of progressive design allows for the evolution of only those parts required for the task in a more flexible design.

In the case of the Khepera robot, when results obtained in the evolution of the eleven modules in one single stage process are compared with those obtained by three stages, it is observed that the progressive design of the controllers obtained a slightly better mean fitness value than the mean fitness obtained in the single stage case. Furthermore, the multiple-staged approach generated a valid solution 100% of the time, with the one-stage process coming in at only made 90% of the time. For that particular case, progressive design proved to be more stable at finding good controllers than the single stage process. One reason is that progressive design performs the evolution in small steps and in reduced searching spaces, building new solutions in new stages beginning with an already stable solution provided by the previous stage.

The negative side to this approach is that only a good enough solution can be provided. Due to the fact that previously evolved modules are frozen from evolving in the new stages, new stages have to implement the solutions found in previous stages.

To sum up, it should be noted that it is not altogether clear whether the progressive design method will be useful in more and increasingly complex robots with hundreds of modules. Even though progressive design allows the evolution of just a few modules in one stage, in the case of hundreds of modules the last modules to be evolved will have hundreds of connections to evolve during that stage, which will result in the search space being large again. Whether the solution found at that moment will be able to lead the new stage towards a point in the fitness landscape where a solution is near will have to be analyzed in future work. In both the Khepera and the Aibo experiments, it was observed that the solutions for one stage rapidly evolved from the solutions found in previous stage, manifesting this *good landscape starting point* effect. It would appear that the method will be valid for more complex agents, if a progressive enough strategy is performed.

6.6 Conclusions

The progressive design method has been described for the generation of controllers for complex tasks in complex robots. In this methodology, modularity is created at the level of the robot device by creating an independent neural

module around each of the robot's sensors and actuators. This small conceptual modification from functional modularization is the reason of the reduction of both the search space dimension and of the bootstrap problem by allowing for the separate evolution of each device in stages. The architecture hence performs modularization in the organization as well as in the learning.

This special type of staged evolution evolves the neural controller in stages, using evaluation-tasks which are conditioned for the devices to be evolved. It has been stressed that determining the evaluation-tasks and the set of modules to evolve in each stage is the designer's job, and that no general formula exists. In general, the designer's knowledge of the problem will play a relevant role, thereby introducing bias into the evolutionary process, which appears unavoidable in complex environments.

The results presented show how a distributed architecture is able to generate a controller for the complex task of walking in a complex quadruped robot with 12 degrees of freedom by using evolutionary robotics. It was shown how the direct evolution of that behaviour was not possible due to the size of the search space for the evolutionary algorithm.

Woman's influence is powerful, especially when she wants something

Josh Billings, American Humorist

7 Adding external influence

This chapter describes how a progressively designed controller can slightly modify its behaviour based on the tonic value of a signal external to the controller. This feature can be useful to tune, or even change, the current behaviour of the controller once it has been completely evolved and installed on the robot. This chapter describes how to integrate this feature into a DAIR controller. An application to the Aibo robot is shown.

7.1 Description

In this chapter we will describe a method to generate a DAIR neural controller for a complex robot, whose behaviour can be tuned based on a tonic value from an external source. The signal modifying the behaviour will be referred to as the *TOC* (from *TOnic Control*). The TOC signal is intended to originate from outside the DAIR controller, and could potentially stem from a user control panel or any other external deliberative modules which may have been implemented with other mechanisms different to DAIR. What is mandatory is that the TOC signal must at every time step provide a value which *represents* a desired status, from the outside module, and into the behaviour of the controller. The procedure described here is based on a similar idea applied in [Ijspeert, 1998], where the speed and direction of a swimming and running salamander is modified depending on the value of a tonic signal[1]. TOC signal values modified the oscillation pattern of the salamander controller, enabling the salamander to walk, turn and swim.

The main aim of developing such method was to allow for the interconnection between a DAIR controller, - which is basically reactive, and knowledge insertion from a superior deliberative layer, which should control and modify the DAIR controller behaviour according to its deliberations. The mechanism implemented here is not intended to cause a complete change of the controller's behaviour: let's say from following a wall to looking for food. Instead, the

[1] In fact, the work described in this chapter was performed while residing at the BIRG laboratory at EPFL under supervision of its director, Professor Auke J. Ijspeert and Cyberbotics CEO, Dr. Olivier Michel.

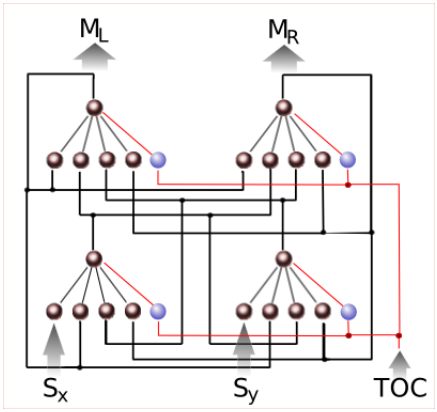


Figure 7.1: Schematics of a final DAIR controller with TOC signal, for the control of a robot with two sensors and two actuators.

objective is to slightly modify the currently evolved behaviour by providing an additional adaptation step. The behaviour will be the same, albeit in a different *tone*.

An example is an *obstacle avoidance* behaviour for a robot. For a robot equipped with this behaviour a potentiometer could provide the TOC signal's value. This value could be used to indicate the distance from obstacles at which to start avoiding them. Another more complex example is the one presented in section 7.3, where the tonic signal is used to change the speed of the Aibo robot's walking pattern. In this case, the TOC value is used to indicate to the IHU's the frequency at which they must oscillate.

The addition of the TOC signal to a DAIR controller must be performed during the generation of the DAIR controller. It is not possible to add such control once the whole DAIR controller has already been created, because TOC based control has to be embedded in the controller to be able to properly affect the behaviour. TOC signals therefore need to be included in each of the stages of the progressive design of the controller for progressively designed controllers. Exceptions can be made with single staged simple controllers. In such cases the DAIR controller can be evolved first, and the TOC signal added afterwards.

The use of a TOC signal to control a DAIR controller does not imply that the same TOC value will be used for all of the DAIR controller's modules at the same time. It is indeed possible to use different TOC signals for different groups of the controller's modules, hence generating multiple behaviours from a single controller. The method described here does not show any limitation about how different TOCs are used for different modules. However, this research has been limited to the case of all the modules sharing a common TOC.

7.2 Procedure

The procedure begins with an already evolved controller at a certain stage. Let's imagine, for the sake of simplicity, that the desired controller only needs a single stage, such as the case of the contour-following behaviour or the garbage collector; the DAIR methodology was applied, and a distributed controller was obtained which performs the desired behaviour in the robot. In this case, the requirement is to modify the controller's currently evolved behaviour, based on the TOC signal's value.

Basically, the method consists of, first, adding a new connection (the TOC input) to the inputs of each module, and then measuring behavioural differences in the controller when the value of the TOC signal changes. Finally, to evolve the connections again, based on the difference in behaviour that best suits the difference in TOC signal value.

During the evolutionary process the robot's behaviour is measured, that is, tested with different TOC values. Controllers accomplishing the behaviour change desired by the designer will obtain better fitness values, and are hence evolved forward. The result is a DAIR controller which changes its behaviour according to the value of the TOC signal.

Differences in the behaviour will be measured by providing a *behavior measurement function*, noted as *behaviour_measure*. This function, defined by the designer of the controller, provides a measure of the behaviour accomplishment, based on the features the designer means of the TOC signal. For example, to change speed based on the TOC signal value, a *behaviour_measure* function may measure the robot peak speed, an average over a period of time; another option may be to measure only its value when going straight. It depends on the global task the robot has to solve, and is up to the designer to decide. Once the *behaviour_measure* function has been defined, differences will be provided by comparing their respective values in different runs with the given controller, where each run will test the same controller with different TOC values.

The procedure begins in the same way as in the progressive design method described in chapter 6. Firstly, a staged evolutionary strategy is selected by the designer. Hence the number of stages N required to evolve the controller is decided, as well as the modules that will be evolved in each stage. At this point, the first evolutionary stage is performed in the same way as described in chapter 6, using the provided f_1 fitness function. Once the modules involved in stage-1 have been evolved and the stage finished, a new input t is added to all the IHU modules evolved. That new input will be the input from the TOC signal. The weight connecting this new input to the inner neurons of the neural network of each IHU is randomly initialised. The evolutionary process from stage 1 is then resumed, but using a modified fitness function f_1' which adds an additional term to the original fitness function, indicating how the controller's behavior has to change based on the TOC value. The new fitness function will now be something different, including the f_1 term, and an additional term based on the

$behaviour\_measure_1$ value,

$$fitness = f_1' \cdot g\left(behaviour\_measure_1\right)$$

During the evolutionary process with f_1', the evaluation of a given controller is performed several times, with different TOC values:

1. In the first evaluation, the TOC signal value is set to 0 for all the evaluation steps. Once the evaluation for that controller is finished, f_1 fitness is calculated as well as the $behaviour\_measure_1$ for that evaluation. They will be called f_{11} and $behaviour\_measure_{11}$. The first sub-index indicates the current stage being evolved. The second index represents the current evaluation.

2. The same controller is then evaluated again, but increasing the tonic signal value by a certain ΔTOC amount. The amount to increase depends on the application, and must be decided by the designer.

3. The new value of the TOC signal will affect the controller's behaviour, and so a new and different f_{12} fitness value, as well as a $behaviour\_measure_{12}$ must therefore be calculated again after the evaluation.

4. If the robot's behaviour is still acceptable, that is,

   $$f_{12} \geq \theta$$

   $$behaviour\_measure_{12} > behaviour\_measure_1$$

   the TOC value is incremented again by ΔTOC and a new evaluation is performed. A fitness value below a given θ threshold means that the behaviour produced by the controller has downgraded too much due to the influence of the TOC value. At this point, our consideration is that the robot does not perform the required behaviour anymore, and the controller's evaluation is finished. The value for θ is determined by the designer.

5. The evaluation process will be repeated n times until either the f_{1n} fitness value decreases to below a certain value, or the $behaviour\_measure_{1n} \leq behaviour\_measure_{1(n-1)}$.

6. Finally, the last fitness for the controller is calculated based on all the fitness f_{1i} with $0 < i < n$ and the number of times that the TOC signal successfully produced a good behaviour, that is $n - 1$. It is up to the designer how to integrate this information into a single f_1' fitness function, but it can be performed by multiplying the last successful fitness function by the (normalized) number of times that the TOC signal was changed,

   $$f_1' = f_{1(n-1)} \cdot \frac{n-1}{maxNumChanges}$$

where $maxNumChanges$ is the maximum number of changes that the TOC signal can theoretically perform. If the TOC value changes from 0 to 1, this value is,

$$maxNumChanges = \frac{1}{\Delta TOC}$$

7. The f_1' fitness is the total fitness assigned to the controller. The general evolutionary process is repeated until a suitable controller for the first stage is obtained.

8. When the first stage of the controller is ended, the second one is started, where new modules are added to the controller. For this second stage, the TOC value of both new and first stage modules is set to zero.

9. Newly added modules are then evolved as explained in chapter 6, using a f_2 fitness function. When the level of proficiency reaches the desired value the evolution of the controller with TOC influence is started. A new $behaviour\_measure_2$ is determined.

10. The TOC evaluation steps described above are now repeated for this new stage, where the TOC values do not affect the whole controller, but only the newly added modules, taking f_2' as evaluation criteria.

11. The whole process is repeated for all the designed evolutionary stages, by defining at each stage functions f_i' f_n, and $behavior\_measure_i$. The final result is the complete DAIR controller whose behaviour changes depending on the TOC value.

An example of the application of this process is shown in the next section where an Aibo robot evolves a different velocity in its walking behaviour.

7.3 Example: influencing the Aibo walking

This section shows in one complex example how to influence a DAIR controller through a TOC signal. The controller is the one developed in section 6.4, and performs a walking behavior in the Aibo robot. For this example, it is requested that the robot be required to change its walking speed based on a TOC value, which basically implies the modification of the frequency at which joints oscillate. The velocity change should of course be synchronised between all the modules.

The generation of the TOC version of the controller will require the repetition of all the stages developed in the 6.4 section, following the method described in section 7.2. Hence, the controller is generated in the four different stages which follow. The evolutionary process is the same as the one described in the 6.4 section, and uses the same parameters. The following sections only describe the additional process required to evolve the walking behaviour at a different

speed. Please refer to section 6.4 for additional information on any parameter or method.

In order to simplify the generation of the controller and speed up the process, different evolutionary stages will not be performed for each type of joint. Instead, at every stage, only the controllers related to the J1 joint type will be evolved and then copied to the other types of joints. Hence, stage 4, coupling between different types of joints, will evolve the couplings between joint types adapting the copied controllers to their specific joint types. It was found that this process was good enough to produce the expected results, even if a better suited controller might have been possible if an independent controller were created for each joint.

7.3.1 First stage: single joint oscillation at several speeds

As described in section 6.4.5.2, the DAIR controller is only composed of two modules at this stage: one for the joint sensor and another for the joint motor. The fitness function to obtain an oscillation was the 6.5 equation, composed of two parts: joint variance and number of oscillations. Only the part measuring the variance, equation 6.4, will be used now, and the part measuring the number of oscillations will vary depending on the TOC signal value. It was empirically observed that a minimum value of 0.5 should be obtained to assert that the joint is oscillating, $\theta_1 = 0.5 f_{1k} > 0.5$.

The $behaviour\_measure_{1k}$ function for stage-1, k-th evaluation, was defined measuring the number of crossings through the mean oscillation position of the joint. Finally, the final fitness function was defined as follows,

$$fitness_1 = f_{1(n-1)} \cdot behaviour\_measure_{1(n-1)} \cdot \frac{n-1}{maxNumChanges}$$

where $n-1$ is the last successful evaluation of the controller. By default, TOC signal values will remain between 0 and 5 throughout all the stages. The increase in TOC signal for each evaluation will be of $\Delta TOC = 0.1$, so

$$maxNumChanges = \frac{5}{\triangle TOC} = 50$$

Based on the described setup, a large range of variable speeds was obtained for joint J1. Table 7.1 shows the number of generations used and the maximum fitness obtained, and summarizes the oscillation ranges obtained for each type of joint. The range is quite large and linear, where small changes of the TOC signal produce small changes in the speed (frequency). This linearity is not mandatory, and indeed can have strange non-linear factors for other applications, as was indicated in [Ijspeert, 1998]. Figure 7.2 shows how the oscillation frequency changes when the tonic value increases[2]. Due to the maximum velocity of the joints, a reduction of the oscillation amplitude is observed as the frequency of oscillation increases.

[2]Videos showing the robot behaviour in the simulator are available at www.ouroboros.org/thesis

Table 7.1: Table of oscillation speed ranges obtained for the J1 joint type when evolving a single joint.

Joint	TOC range	Freq. range (Hz)	N. gen.	Max. fitness
J1	0.0 - 1.4 (15 values)	0.2 - 0.7	40	0.75

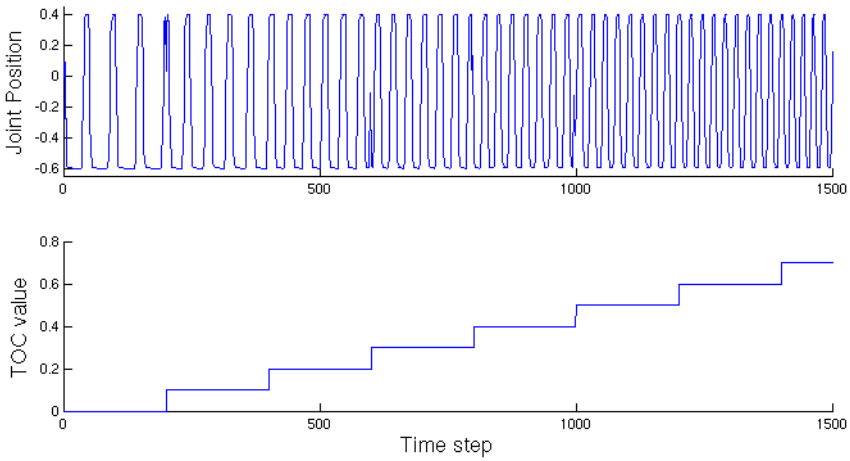


Figure 7.2: In this figure, the controller obtained for one single joint is tested (on joint J1-LF). The oscillatory pattern obtained shows how the oscillation frequency increases along the TOC value. The change in frequency value is very smooth.

Table 7.2: Table of oscillation speed ranges obtained for the J1 joint type when evolving two joints in counter phase.

Joint	TOC range	Freq. range (Hz)	N. gen.	Max. fitness
J1	0.0 - 0.7 (8 values)	0.31 - 0.7	40	0.85

7.3.2 Second stage: two joints oscillating at several speeds

The goal for this stage is to obtain two joints oscillating in counter phase. Oscillation frequency changes in both joints must be similar to maintain phase between them. The controller is now composed of four IHU modules. The same trick described in section 6.4.5.3 is used to simplify the evolutionary process. That is, the controller developed in the previous stage is duplicated in this stage to control the two joints, and only the connections between the two are evolved. For the fitness function which decides whether the behaviour is or is not a counter oscillatory one, the fit_var term in equation 6.6 is used, $f_{2k} = fit\_var$. Again, it is empirically observed that a minimum value of 0.5 has to be obtained in order to assert that the joint is oscillating.

The $behaviour\_measure_{2k}$ function is defined as the number of crossings through the mean position value of the joint, and it is calculated independently for each joint, A and B,

$$behaviour\_measure_{2k} = \left\{ fit\_cross_k^A, fit\_cross_k^B \right\}$$

Finally, the final fitness function for the controller will be calculated as follows,

$$fitness_2 = fit\_cross_{n-1}^A \cdot fit\_cross_{n-1}^B \cdot fit\_var_{n-1} \cdot \frac{n-1}{maxNumChanges}$$

Table 7.2 shows the number of generations used and the maximum fitness obtained, and summarizes the oscillation ranges obtained for each type of joint. Figure 7.3 shows how the oscillation frequency changes when the tonic value increases. A small reduction in the frequency range that the controller is able to oscillate in whilst remaining in a coordinated oscillation is observed from the previous stage to this one.

7.3.3 Third stage: four joints oscillating at several speeds

This stage coordinates oscillation between four joints of the same type, and synchronizes them when changing speed. As in the previous section, the controller obtained in the previous stage is duplicated to control from two to four joints, and only the connections between the modules are evolved. For the fitness function determining whether the behaviour is or is not a counter oscillatory one, the two last terms in the 6.7 equation are used, (fit_oscil) measuring how good the oscillation of one of the legs is (the left fore), and (fit_phases) evaluating how good the phase differences are between this leg and the others for a walking pattern,

$$f_{3k} = fit\_oscil_k^{\mathrm{LF}} \cdot fit\_phases_k$$

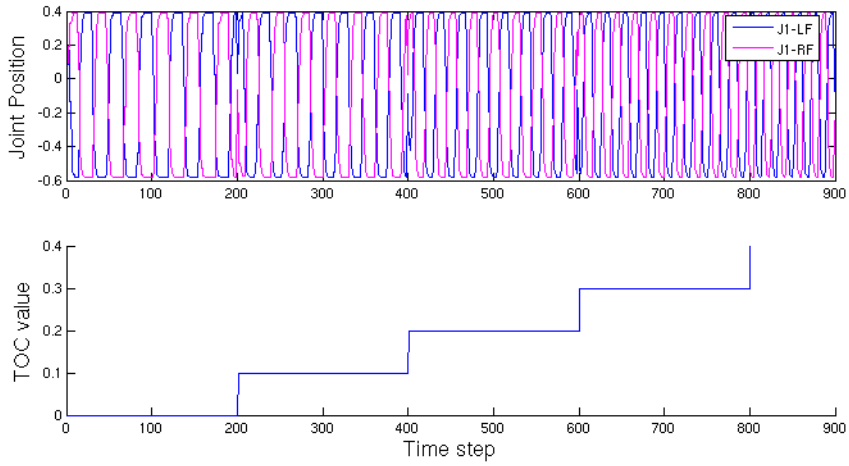


Figure 7.3: Oscillation patterns obtained for each type of joint, when two joints are evolved.

where

$$fit\_phases_k = fit\_phases_k^{\text{LF-LR}} \cdot fit\_phases_k^{\text{LF-RF}} \cdot fit\_phases_k^{\text{LR-RR}}$$

The threshold values actually observed in order to obtain a real oscillation are as follows:

$$\begin{gathered} fit\_oscils_k^{\text{LF}} > 0.7 \\ fit\_phases_k^{\text{LF-LR}} > 0.49 \\ fit\_phases_k^{\text{LF-RF}} > 0.49 \\ fit\_phases_k^{\text{LR-RR}} > 0.49 \end{gathered}$$

It is observed that the threshold for $fit\_oscil_k^{\text{LF}}$ increased from previous stages value (0.5) to a higher value (0.7). This is due to the fact that a higher quality oscillation for the left fore leg is required because all the other legs will synchronize their oscillation to that leg.

The $behaviour\_measure_{3k}$ is defined by the number of crossings each joint performs. Hence, it is composed of four terms: one for measuring the crossings of each joint,

$$behaviour\_measure_{3k} = \left\{ fit\_cross_k^{\text{LF}}, fit\_cross_k^{\text{LR}}, fit\_cross_k^{\text{RF}}, fit\_cross_k^{\text{RR}} \right\}$$

Evaluations with an increased TOC signal value are performed as long as all four $fit\_cross_k^{(\cdot)}$ values increase from the previous evaluation to the current one. Finally, the overall fitness assigned to a controller is calculated as follows,

$$\begin{aligned} fitness_3 \quad = \quad & fit\_cross_{n-1}^{\text{LF}} \cdot fit\_cross_{n-1}^{LR} \cdot fit\_cross_{n-1}^{\text{RF}} \cdot fit\_cross_{n-1}^{RR} \cdot \\ & fit\_oscil_{\text{n-1}}^{\text{LF}} \cdot fit\_phases_{\text{n}} \cdot \frac{n-1}{maxNumChanges} \end{aligned}$$

Table 7.3: Table of oscillation speed ranges obtained for the J1 joint type, when evolving four joints in walking style phase.

Joint	TOC range	Freq. range (Hz)	N. gen.	Max. fitness
J1	0.0 - 0.7 (8 values)	0.46 - 0.7	40	0.54

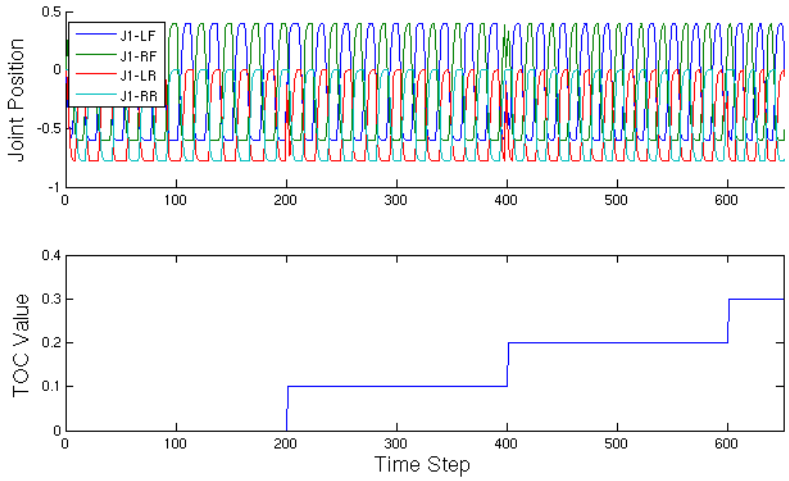


Figure 7.4: Oscillation patterns obtained for each joint type, when four joints are evolved.

Table 7.3 shows the number of generations used and the maximum fitness obtained, and summarizes the oscillation ranges obtained for each joint type. Figure 7.4 shows how the oscillation frequency changes when the tonic value increases.

7.3.4 Fourth stage: coupling the three layers

From the previous stage, a layer of J1 joints of the same type which oscillate with a walking gait phase relationship between joints is available, and whose oscillation speed can be changed. This stage replicates the obtained layer to the other two: J2 and J3, and coordinates the oscillation between the three different layers by evolving the connections between them. The three layers must be able to change speed based on the TOC value.

For the fitness function determining the counter oscillatory behaviour, two terms are used again: fit_oscil scoring the oscillation of the left fore joint of each layer, and fit_phases for the phase differences of this leg and the others for a walking pattern, for each of the three layers,

$$f_{4k} = fit\_oscil_k \cdot fit\_phases_k$$

Table 7.4: Table of oscillation speed ranges obtained when evolving four joints in walking style phase.

TOC range	Freq. range (Hz)	Num. gen.	Max. fitness
0.0-0.1 (2 values)	0.5 - 0.503	175	0.12

where

$$\begin{aligned} fit\_oscil_k = &\quad fit\_oscil_k^{\text{J1-LF}} \cdot fit\_oscil_k^{\text{J2-LF}} \cdot fit\_oscil_k^{\text{J3-LF}} \\ fit\_phases_k = &\quad fit\_phases_k^{\text{J1}} \cdot fit\_phases_k^{\text{J2}} \cdot fit\_phases_n^{\text{J3}} \\ fit\_phases_k^{\text{Jx}} = &\quad fit\_phases_k^{\text{Jx-LF-LR}} \cdot fit\_phases_k^{\text{Jx-LF-RF}} \cdot fit\_phases_k^{\text{Jx-LR-RR}} \end{aligned}$$

and the threshold values actually observed are as follows,

$$\begin{aligned} fit\_oscils_k^{\text{Jx-LF}} &> 0.7 \\ fit\_phases_k^{\text{Jx-LF-LR}} &> 0.49 \\ fit\_phases_k^{\text{Jx-LF-RF}} &> 0.49 \\ fit\_phases_k^{\text{Jx-LR-RR}} &> 0.49 \end{aligned}$$

where Jx stands for J1, J2 and J3 values.

The $behaviour\_measure_{3k}$ is defined by the number of crossings each joint performs. Hence, it is composed of four terms, one for measuring the crossings of each joint,

$$behaviour\_measure_{3k} = \left\{ fit\_cross_k^{\text{Jx-\{LF,LR,RF,RR\}}} \right\}$$

Finally, the overall fitness assigned to a controller is calculated as follows:

$$\begin{aligned} fitness_3 = &\quad fit\_cross_{n-1}^{\text{LF}} \cdot fit\_cross_{n-1}^{LR} \cdot fit\_cross_{n-1}^{\text{RF}} \cdot fit\_cross_{n-1}^{\text{RR}} \cdot \\ &\quad fit\_oscil_{n-1} \cdot fit\_phases_n \cdot \frac{n-1}{maxNumChanges} \end{aligned}$$

where

$$\begin{aligned} fit\_cross_{n-1} = &\quad fit\_cross_{n-1}^{\text{J1}} \cdot fit\_cross_{n-1}^{\text{J2}} \cdot fit\_cross_{n-1}^{\text{J3}} \\ fit\_cross_{n-1}^{Jx} = &\quad fit\_cross_{n-1}^{\text{Jx-LF}} \cdot fit\_cross_{n-1}^{\text{Jx-LR}} \cdot fit\_cross_{n-1}^{\text{Jx-RF}} \cdot fit\_cross_{n-1}^{\text{Jx-RR}} \end{aligned}$$

Table 7.4 shows the number of generations used and the maximum fitness obtained, and summarizes the oscillation ranges obtained. In the same way as in the case of two joints, it is observed a huge reduction is observed in the frequency range that the controller is able to oscillate in while maintaining a coordinated oscillation.

7.3.5 Discussion

The results obtained show how the different joints of the same type can vary their speed based on the TOC signal value. The change in speed of the whole Aibo robot is not very significant though. There could be several reasons for

this, but the most likely is that too many new connections to be evolved are involved in the last stage. On the other hand, the proposed staged evolution is only one of many possibilities using the DAIR. Fitness functions can be improved in several ways, leading to other sub-optimal (and probably successful) results. For instance, one example is that the fourth stage could be divided into two stages, one where modules for joints J1 and J2 are connected using a fitness function based on oscillation patterns, and a second one where the J3 layer of controllers is added, and the fitness function is only related to walking distance. In any case, this is just an example and other ways of evolving it may be found.

Once the complete walking controller with velocity change is obtained, the robot can be controlled with different TOC values for different parts of the controller. This means that a certain TOC value can be applied to the modules of the left-hand side of the robot, and a different TOC value applied to the modules of the right-hand side. This may lead to a turning behaviour with one single controller, in a similar way as applied in [Ijspeert, 1998].

As a summary of this section, we must point out that even if the final result was not totally successful (just two TOC values are allowed to change Aibo's speed), the section demonstrates an application of how an external signal can affect a DAIR controller. The main drawback of the method is that it requires considerably more time and external knowledge to be introduced.

7.4 Conclusions

This section has demonstrated how to include an external signal inside a DAIR controller which allows the modification of the evolved controller's behaviour. It only shows proof of the method's concept when applied to a complex robot. Hence, several issues remain open for further study: such as the reason why the TOC range decreased from one stage to another, whether the change in behaviour is lineal with the change in TOC values, or whether a better strategy would have lead to a larger range of speeds in the robot. The use of such a procedure to modify the walking trajectory of the robot by applying different TOC values to different modules at the same time is left for the future.

The meaning of a experience is integrated in the experience itself. Is part of it, is embedded into it, and has no meaning outside that concrete experience

Oscar Vilarroya (The dissolution of mind)

8 DAIR inner analysis

In previous chapters it was shown how a *society* of distributed IHU's was able to control different types of robots while performing some behaviour.

This chapter analyzes how IHU's deal with information when they are working on the generation of a behaviour for a robot. What the role of each IHU is in the final robot behaviour will be analyzed, as well as how the IHU's organize and collaborate between each other in order to generate the final global behaviour. By analyzing the distributed ensemble of IHU's it is observed that modules create their own communication values for the cooperation between themselves. Furthermore, it is observed that IHU's always communicate the same value to their peers when the situation that the robot is experiencing is similar, even if the actual sensor values are different. This observation leads to a discussion about how IHU's manage information so that the output of the IHU's is taken as a meaningful *internal representation* of the current situation of the robot. This internal representation has emerged through interaction of the robot with the environment, or, otherwise stated, the robot acquires its own semantics. Hence, what this section basically explains is the addition of the term *Internal Representation* in the name of the (DAIR) architecture.

The chapter begins introducing some notation about the observable vector that allows us to infer the internal representation of the robot. Next, this vector is used to analyze the behaviour in the contour-following experiment, and to provide an automatic way for the user to extract understandable knowledge from the internal representation. Additional results are presented analyzing the garbage collector behaviour, as well as a discussion of the Aibo behaviours obtained in previous chapters.

8.1 Introduction

In previous chapters, the DAIR architecture was shown generating the required behaviours for robot control in various different tasks. In order to achieve this, a cooperation/coordination between all the IHU's was obtained through an evolutionary process. This coordination lead to a given robot behaviour, with each IHU having a particular view of the robot's current situation. It is important to

show how this cooperation works, and what the final result of this cooperation consisted of.

It will be observed that the IHU elements generate a cooperative behaviour between each other. They communicate their status to the other IHU's through their output, in the robot's different situations. This kind of *communication mechanism* leads to an internal representation of the robot situation, where the status of all the IHU's are taken into account in any given situation. The internal representation can be observed as a semantic representation of the robot's current situation, with the robot associating meaning to its experiences. The generation of this meaningful situation permits the robot to categorize its situation by means of a state vector which is easily accessible from the outside even if neural networks have been used.

8.2 Definitions

Some terms will be defined before going further in the analysis of the IHU behaviour. It is important to notice that the terms defined here only apply for completely evolved controllers. They are not used for controllers that are still in the process of evolution.

For the analysis of the IHU's it will be observed how the outputs of the IHU modules change in different situations *experienced* by the robot while it is performing a task that it has been evolved for. For this purpose, the *state vector* is defined as the vector that contains the output values of all the robot's IHU's at any given time. That is, for a robot controller composed of N sensors and M actuators, the state vector is defined as,

$$state\ vector(t) = (O_{S1}(t), O_{S2}(t), \ldots, O_{SN}(t), O_{A1}(t), O_{A2}(t), \ldots, O_{AM}(t))$$

The vector can be plot over a period of time and observe how it changes in the different situations the robot is experiencing. For instance, figure 8.1 shows the plotting of the state vector for a Khepera II robot while controlled by 4 IHU's performing a contour-following behaviour. Each coloured line shows the current value of a IHU's output. The state vector will help to investigate whether any correlation exists between the current state of the IHU's and the situation that the robot is experiencing.

Now, the *internal state* of the robot is defined as the value of the state vector at a given time. Thus the internal state of the robot is codified as a vector of n dimensions, where $n = M + N$ is the number of IHU's. The interesting point is to observe how the internal state changes while the robot performs its evolved task, and identify any correlation between its behaviour and the current internal state. In fact, as we will see below, the internal state coherently changes as the robot situation changes. This will lead us to show that the internal state actually represents how the robot perceives the current situation that it is experiencing.

The internal state actually remains relatively stable around a set of values when the robot is involved in similar situations. The *model vector* is defined as the set of values of the state vector related to one given internal situation,

and hence represents the given situation experienced by the robot. As it will be shown analyzing the state vector in two examples, the DAIR architecture will identify all the different situations where the robot is involved while running its behavior. The identification of such situations will be expressed by maintaining a given value of the state vector. This value will remain stable for similar situations, and will indicate the internal state of the robot. This is the model vector. Robot situations are associated to the robot experience and they can correspond to recognizable human meanings, though this is not mandatory.

8.3 Contour-following behaviour state vector analysis

The IHU's knowledge management will be analyzed within a successfully evolved DAIR controller through the contour-following behaviour example. The state vector of the controller created in chapter 4 for the generation of the orbiting behaviour on a Khepera II robot will be studied. In this case, the state vector is composed of only four components, corresponding to the output of the two sensors and two actuators,

$$state\ vector(t) = (O_{Sx}(t), O_{Sy}(t), O_{Mr}(t), O_{Ml}(t))$$

where $O_{S(\cdot)}(t)$ is the output from the IHU associated to the sensor $S_{(\cdot)}$ (IHU-$S_{(\cdot)}$) delivered to the other IHU's, and $O_{M(\cdot)}(t)$ is the IHU's output of the $M_{(\cdot)}$ (IHU-$M_{(\cdot)}$) motor. The group of neural modules used to control the robot in this test was the one that obtained the best performance in the ten evolutionary tests performed in chapter 4.

8.3.1 State vector of the contour-following behaviour

The state vector trajectory for this robot during a typical operation is shown in figure 8.1. The robot starts in a free space with it's sensors not detecting anything and it initiates a circular movement, indicated by the activation of the IHU-M_l and IHU-M_r at different power levels. The execution of this mechanism ensures that the robot will encounter an obstacle at some point on its circular trajectory; either the central object or the wall. The radius of the performed circle depends on the conditions in which the robot was evolved, basically the distance-to-obstacle at the beginning of the evolution.

Once the robot encounters the obstacle with sensor S_y, the state vector changes for the motor IHU's, producing an opposite rotation direction. Also, the part of the sensor of the state vector changes to indicate that the robot is experiencing a different situation. The rotating behaviour in the opposite direction is activated, until the S_y sensor detects nothing and sensor S_x reaches a certain level[1]. At this point, the robot is, more or less, aligned with the object.

[1]Sensor values shown on the plots of this chapter increase their value when the distance detected decreases. This is how the simulator works.

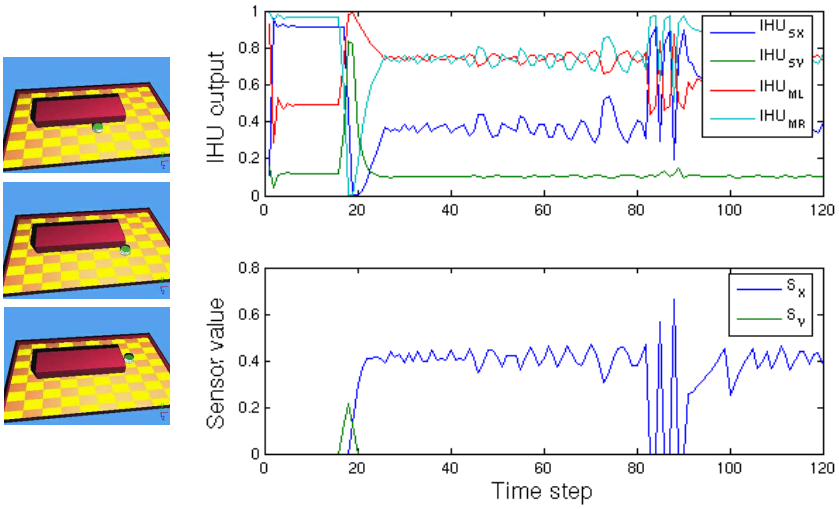


Figure 8.1: This plot shows a typical run of the contour-following behaviour. Left: Sequence of robot movements. Right-top: state vector plot. Right-bottom: distances detected by sensors during the run.

Again, the state of the vector changes by applying a similar velocity to both motors so that the robot moves along the object. After a while, in step 83, the robot loses contact with the object because it has reached a corner. The IHU's quickly change their state to activate again the counterclockwise rotation procedure. Once the robot detects the object again with sensor S_x, the *move along object* state is again activated.

8.3.2 Identified model vectors (states)

By visual inspection of the robot changing behaviour, four different states are identified: the robot detects nothing; the robot detects something with sensor S_x; the robot detects something with sensor S_y; and the robot detects something with both sensors. We will inspect below the state vector looking for any model vector uniquely associated to those states, and hence, associate those model vectors to the current situation (state) of the robot.

In order to find model vectors associated to the robot state, to ascertain how stable they are, and which their value ranges could be, the state vector is measured on each of the particular situations described above following this experimentation: one of the four described situations is artificially presented to the robot and its vector state evolution towards a stable state is observed. In that artificial situation, the robot is not allowed to change its status by acting upon the motors, because this would change the situation that the robot experiences. So, the output generated by the IHU's of the motors is not effectively

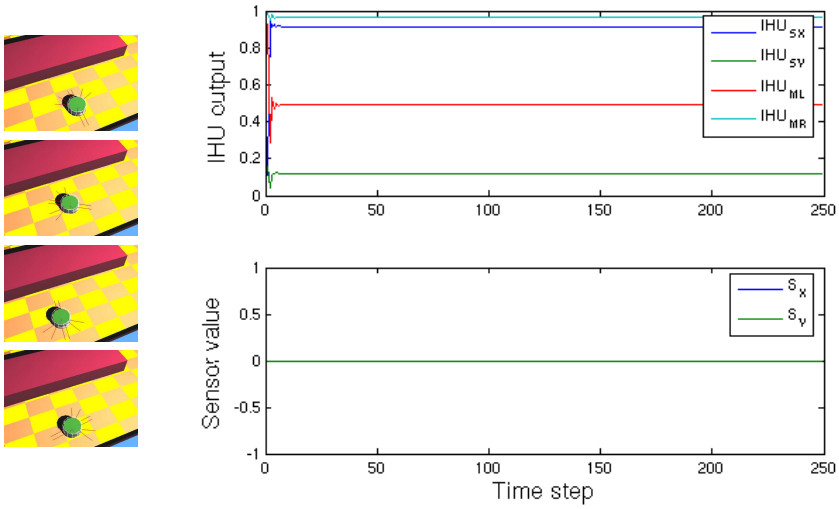


Figure 8.2: Left: sequence of movements of the contour-following robot when it does not detect anything and moves in counterclockwise circles. Right: vector state in that situation.

sent to the motors for acting upon them, but the signal is sent to the other IHU's. Once the state vector has stabilized, the conditions of that particular state are manually modified so slightly that no change in the state is observed. Distances detected by the sensors are slightly varied by manually moving the robot to a convenient distance. As long as the state vector remains stable, it will be concluded that the current range of values of the state vector form a model vector which defines the situation as experienced by the robot.

The four different situations, corresponding to the states observed, are presented:

State-a Robot in a free space. The robot detects nothing with its sensors. In this case, the robot is placed in the environment where nothing is detected and it is allowed to move around freely while not approaching any object. The state vector evolution obtained is the one depicted in figure 8.2. After a short transient, the state vector becomes stable with a value,

$$state\ vector(t) = (0.9, 0.12, 0.49, 0.96)$$

This stabilized value is identified as the *model vector for State-a*. It will be said that the *meaning* of that model vector is *robot in a free space*.

State-b The robot detecting an object with the S_y sensor. In this situation, the robot detects something in front of it only with the S_y sensor. To observe how the status is activated and at which distance, the robot is situated in

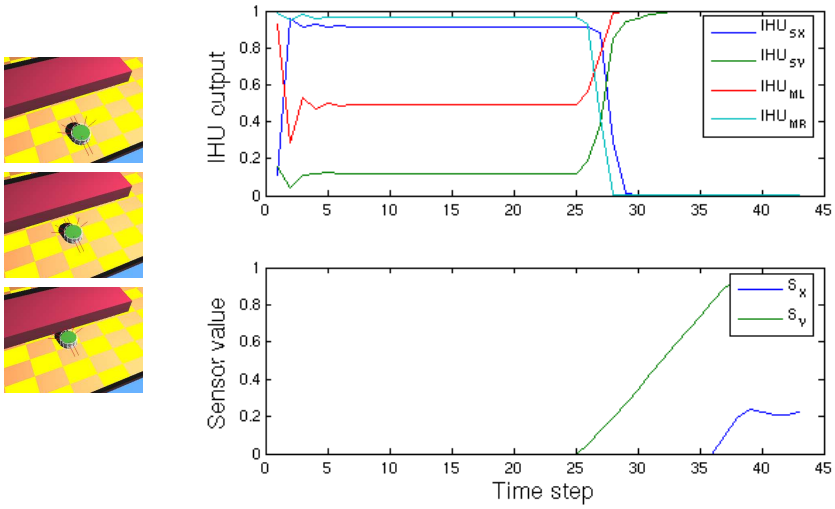


Figure 8.3: Left: sequence of images of the contour-following experiment when the robot approaches the central object from the free space, and detects it with the S_y sensor. Right: vector state transition from State-a to State-b.

a free space and both wheels are blocked from moving towards the central object. The transistion from the State-a to the State-b is clearly observed in figure 8.3: the state is activated after the value of the IHU output associated to the sensor is above a certain threshold, and the state vector remains stable for all the different values of the distance detected by the sensor. This is an important result since it supports the claim that the state vector really is indicating the robot's real state from a conceptual point of view. As a consequence, the controller drives a change in the IHU outputs of the motors. In the figure 8.3, the robot does not actually rotate because the wheels are blocked during the experiment. Hence, the model vector for this state is identified as,

$$state\ vector(t) = (0.0, 0.99, 0.99, 0.0)$$

For the *model vector for State-b*, the *meaning* is *object in front of robot.*

State-c Robot detecting an object with both sensors. In this situation, the robot detects something in front of it with both, S_x and S_y sensors. To test this state, the robot is placed in a free space at an inclination of 45^{o} towards the central object and the wheels blocked from moving the robot towards the object. The transition from State-a to State-c can be observed in figure 8.4. Again, this state is activated when the values are above a certain threshold. It can be assumed that the concept in this case is that

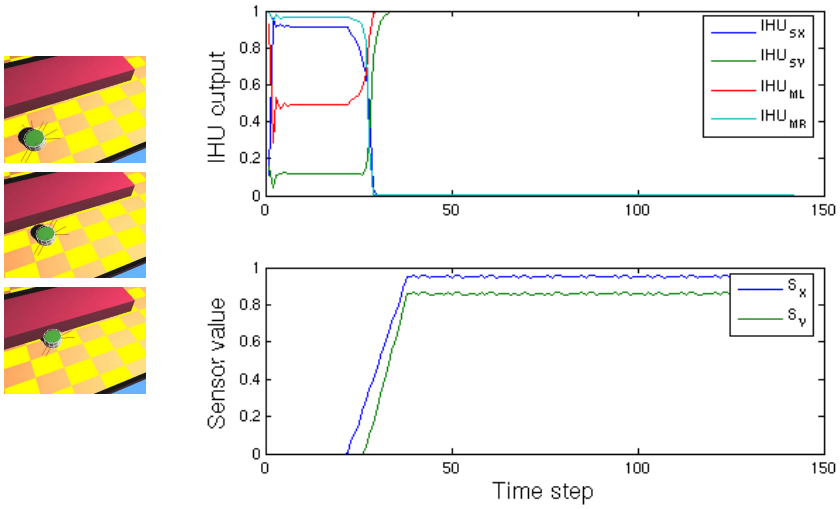


Figure 8.4: Left: sequence of images of the contour-following robot when it does approach the central object from free space, and detects it with both S_x and S_y sensors. Right: vector state in that situation.

there is an obstacle in front of the robot. The model vector for this state is identified as,

$$state\ vector(t) = (0.0, 0.99, 0.99, 0.0)$$

By observing the model vector for State-c, it can be verified that it is exactly the same as for the previous State-b. This means that the robot is identifying something along the lines of *the same situation*, whether the S_x sensor is activated or not. Hence, State-b and State-c both represent the same concept from the controller point of view, related to the task it has to solve, and they require the same actuation, that is, to change rotation direction to clockwise. So, the State-c is discarded as a new state.

State-d Robot moving along the contour of the object. In this case, only the S_x sensor detects something. By placing the robot in close proximity to the object and leaving it free to act with the learned controller, a behaviour and associated state vector as shown in figure 8.5 are obtained. In this case, the robot changes its situation from having the object in front of it to having the object on its left-hand side. The state vector plotted in figure 8.5 shows a non-constant value for the different IHU outputs, apart from IHU-S_y. Values oscillate into a given range resulting in an oscillating behaviour[2]. This oscillation when the robot is moving along the object at

[2]See the video at www.ouroboros.org/thesis

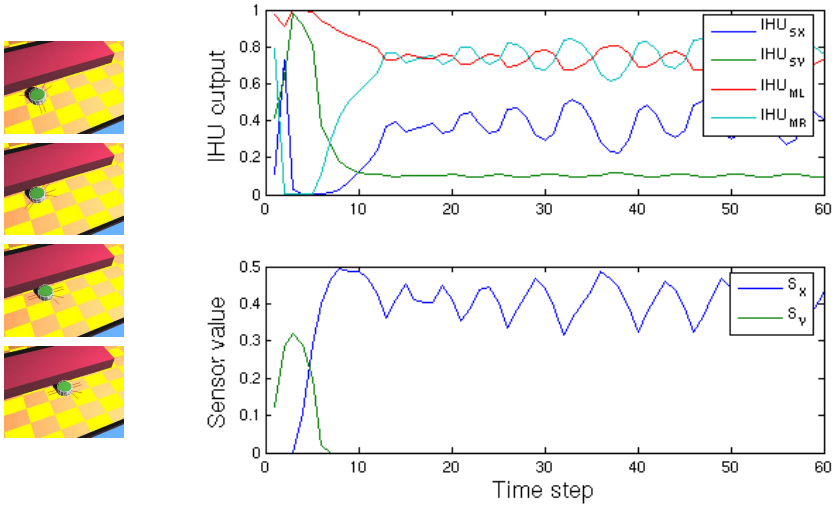


Figure 8.5: Left: sequence of images of the contour-following robot when it changes from detecting the object with S_y sensor, to detecting the central object with the S_x sensor. Right: vector state for that situation.

a CLOSE distance is motivated by the non-constant command controlling the motors, hence sometimes IHU-M_l > IHU-M_r, or inversely. Even if the values move around an average they do not remain stable. The model vector for this state can be described as follows,

$$state\ vector(t) \in [(0.228, 0.1, 0.672, 0.617), (0.51, 0.1, 0.88, 0.839)]$$

or, alternatively,

$$state\ vector(t) = (0.369, 0.1, 0.736, 0.728) \pm (0.141, 0, 0.064, 0.111)$$

The situation depicted in figure 8.5 presents a reduced set of situations, where the couple of situations that the robot faces are limited and do not allow for the observation of how the state of sensor S_x influences in the vector state, or whether its value determines one state or another. To shed light on this subject, we again test the robot moving along the central object at different distances from it. Figure 8.6 shows how the state vector changes for such situations.

During normal operation (figures 8.1 and 8.5) the state vector uses a model vector for moving along the object like the one described by State-d. However, figure 8.6 indicates that State-d is in fact not as stable as observed in figure 8.1 during normal operation. Hence, from figure 8.5, State-d can be divided into three different situations: the robot drifts to the left (when IHU-M_l < IHU-M_r), the robot moves more or less straight forward (when IHU-M_l = IHU-M_r), or the robot drifts to the right (when IHU-M_l > IHU-M_r). These situations

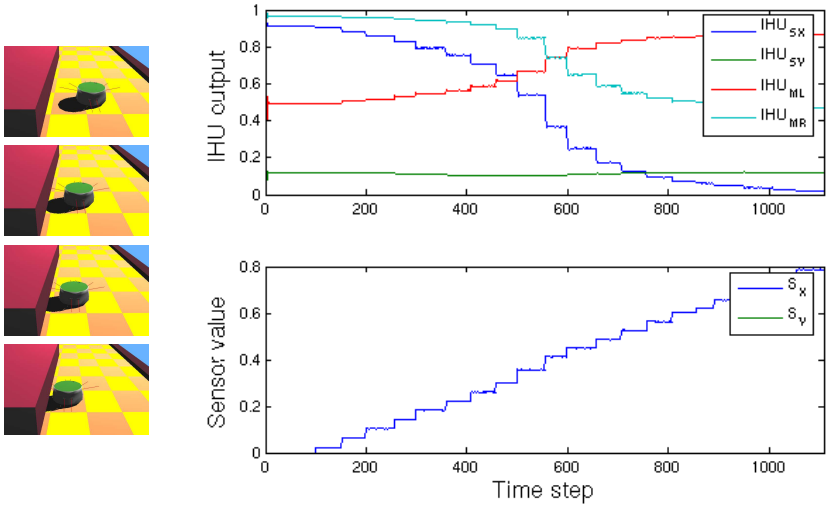


Figure 8.6: Left: sequence of images of the contour-following robot at different distances from the central object. In this situation, only the S_x sensor detects something. Right: vector state for those situations. The state vector contains the measurements at 21 different distances, even though the figure on the left only shows the measurement for four of them.

Table 8.1: Manually extracted model vectors for the contour-following behaviour.

State	IHU S_x	IHU S_y	IHU M_l	IHU M_r
a	0.9	0.12	0.49	0.96
b	0.0	0.99	0.99	0.0
d_1	$0.775 \pm .135$	0.1	$0.55 \pm .06$	$0.925 \pm .035$
d_2	$0.44 \pm .20$	0.1	$0.695 \pm .085$	$0.765 \pm .125$
d_3	$0.135 \pm .105$	0.1	$0.825 \pm .035$	$0.56 \pm .08$

are identified as new states that the robot is *experiencing*, which will be called State-d_1, State-d_2, and State-d_3 respectively. From the data plotted in figure 8.6, states are identified according to the data in table 8.1. In fact, the model vector for d_2 corresponds to the original situation of State-d.

8.3.3 Automatic extraction of model vectors

Up until this point, identification of the model vectors has been manually performed. Even if it worked correctly for that simple contour-following example, manual analysis could not be possible for more complex robots and environments, due to the high dimensionality of the state vector, and the different states that may arise. For this reason, it would be more convenient to generate an automatic procedure that, given a stream of state vector values, determines how many different model vectors exist and under which circumstances. For this purpose, the use of a vector quantizer algorithm called ARAVQ [Linaker and Niklasson, 2000, Bergfeldt and Linåker, 2002] is proposed.

The ARAVQ algorithm works on the IHU's flow of data by identifying different situations coded in. It acts as a moving average along the data flow, capturing model vectors that represent high order concepts, clustering state vectors in the same concept, that is, in the same model vector. This algorithm does not require the number of model vectors to be specified beforehand, but rather generates them automatically.

The algorithm works as follows [Linaker and Niklasson, 2000]: it receives the IHU's current outputs as input at every time step. A finite moving average $\bar{\mathbf{x}}(t)$ is calculated over the last n input signals provided to the algorithm,

$$\bar{\mathbf{x}}(t) = \frac{1}{n} \sum_{i=0}^{n-1} \mathbf{x}(t-i)$$

At each time step, the moving average is compared with all the model vector already created, using a mismatch measure. If it does match with any of the vectors, then no action is taken, and it is assumed that the current input signal corresponds to the matching model vector. Otherwise, the current average is promoted for the generation of a new model vector, meaning that the robot is

experiencing a new situation. The mismatch measure is calculated as,

$$d(V,X) = \frac{1}{\mid X \mid} \sum_{i=1}^{\mid X \mid} min_{1 \leq j \leq \mid V \mid} \{ \parallel \mathbf{x}_i - \mathbf{v}_j \parallel \} \; ; \; \mathbf{x}_i \in X, \mathbf{v}_j \in V$$

where V and X are sets of vectors, and $\parallel . \parallel$ denotes the Euclidean distance.

The promoted average will be a new model vector if two criteria are fulfilled: first, the $d_{\mathrm{M(t)}}$ mismatch between each of the last n inputs and all the $\{\mathrm{M(t)}\}$ model vectors must be greater than a certain δ threshold,

$$d_{\mathrm{M(t)}} = d\left(\{\mathrm{M(t)}\}, \{\mathbf{x}(t), \ldots, \mathbf{x}(t-n+1)\}\right) \geq \delta$$

second, the current moving average must be stable, that means, the $d\bar{\mathbf{x}}(t)$ difference between the $\bar{\mathbf{x}}(t)$ moving average and the last n input signals has to be below a certain stability criteria ε.

$$d\bar{\mathbf{x}}(t) = d\left(\bar{\mathbf{x}}(t), \{\mathbf{x}(t), \ldots, \mathbf{x}(t-n+1)\}\right) \leq \varepsilon$$

Basically, ε is the quantity of noise allowed, in terms of distance between the current average and the models, or variability between models, and δ is the minimum distance allowed between different models. If both criteria's are met, the current average signal $\bar{x}(t)$ is used as a new model vector and incorporated into the current set of model vectors. Additionally, the ARAVQ algorithm incorporates a way to fine-tune the models created. It may occur that the current model vector used for a given concept is not the *best* representation of that concept. Then, the algorithm uses a α learning rate parameter which indicates how a model vector could be modified by new, better instances of it.

This algorithm mainly requires four parameters to be set up: ε, δ, the buffer size n, and α. Their values depend on how large a difference between model vectors is allowed. For the case of the contour-following behaviour, the following parameters were selected:

- For the δ parameter, the minimum for Euclidean distances between all the vector models obtained by hand (see table 8.1) was calculated, and a slightly lower value was used,

$$\begin{aligned} \delta = & \; min\{\parallel x_{Sa} - x_{Sb} \parallel, \; \ldots, \parallel x_{Sd2} - x_{Sd3} \parallel\} \\ = & \; min\{1.705, .142, .54, .926, 1.562, 1.287, 1.072, .398, .786, .1519\} \\ = & \; .142 \end{aligned}$$

- For the ε parameter, the averaged maximum change of the IHU output values for a given state, averaged over all the states, was calculated,

$$\varepsilon = \frac{1}{S} \sum_{i=1}^{S} \frac{1}{M+N} \sum_{j=1}^{S} \frac{IHU_{ij}^{max} - IHU_{ij}^{min}}{2}$$

  where S is the total number of states identified, $M+N$ is the total number of IHU's used, and IHU_{ij}^{max} and IHU_{ij}^{min} are the maximum and minimum

Table 8.2: ARAVQ parameters used for the extraction of model vectors in the contour-following behaviour.

Alpha	Delta	Epsilon	Buffer size
0.03	0.14	0.04	12

Table 8.3: Table of model vectors identified by the ARAVQ algorithm when applied to the state vector of figure 8.2. Association with states has been manually performed.

ARAVQ model	IHU S_x	IHU S_y	IHU M_l	IHU M_r	Manual model
1	0.91	0.12	0.49	0.96	a

output provided by IHU-j when in i-th state. So, from values of the manual analysis,

$$\varepsilon = \frac{(IHU^{max}_{Sx-d1} - IHU^{min}_{Sx-d1}) + \cdots + (IHU^{max}_{Mr-d3} - IHU^{min}_{Mr-d3})}{5 \cdot 4 \cdot 2} = .065$$

- For the buffer size n, it is observed that the state vector stabilizes after transitions lasting no longer than 3 time steps. So a value above that number will be enough to not try to generate model vectors from the transitions. $n = 12$ is selected as the buffer size.
- For the α learning rate a typical value is 0.03.

Hence, the final parameters used in the ARAVQ algorithm are displayed in table 8.2. The application of the algorithm to the different vector states of figures from 8.2 to 8.6, produces the state vectors in tables 8.3 to 8.7.

After applying the ARAVQ algorithm to the IHU's data flow displayed in figure 8.1, three different model vectors were obtained. The model vectors discovered by the algorithm are very similar to the ones manually calculated. The ARAVQ algorithm was able to discover the model vectors, but does not include any information about to which (meaningful) state they actually correspond. However, for this case, it is very easy to manually create the correspondence, by observing either the figures used for the manual identification of the models (figures from 8.2 to 8.6), or by comparing with the manual models summarized in table 8.1.

Table 8.4: Table of model vectors identified by the ARAVQ algorithm when applied to the state vector of figure 8.3. Association with states has been manually performed.

ARAVQ model	IHU S_x	IHU S_y	IHU M_l	IHU M_r	Manual model
1	0.91	0.115	0.48	0.96	a
2	0.0	0.989	0.99	0.0	b

Table 8.5: Table of model vectors identified by the ARAVQ algorithm when applied to the state vector of figure 8.4. Association with states has been manually performed.

ARAVQ model	IHU S_x	IHU S_y	IHU M_l	IHU M_r	Manual model
1	0.91	0.115	0.48	0.96	a
2	0.0	0.961	0.99	0.0	b

Table 8.6: Table of model vectors identified by the ARAVQ algorithm when applied to the state vector of figure 8.5. Association with states has been manually performed.

ARAVQ model	IHU S_x	IHU S_y	IHU M_l	IHU M_r	Manual model
1	0.37	0.1203	0.74	0.74	d

8.3.4 Discussion

In the previous example, four different model vectors were identified for the robot behaviour by using the state vector. A task-related meaning has been manually assigned to those model vectors, because it was possible to visually identify the vector values with a well defined situation of the robot. However, this does not mean that the situation will be the same for other robots or other tasks, where understandable human meanings are assigned to the vector models; that is, it may ocur that the robot creates internal vector states that correspond to a situation interesting (i.e. meaningful) for the robot itself, but where we cannot identify a situation from an external human point of view.

Automatic algorithms for model vectors extraction allow the extraction of model vectors in more complex controllers with dozens of components. In this section, it has been demonstrated that the ARAVQ algorithm is a perfect candidate to extract the model vectors in an automatic way. But, more interestingly, what has been shown is that what the algorithm extracts really corresponds to the states that emerged from the DAIR architecture, and nothing more. Related to the automatic procedure for model vector extraction, it could be argued that the architecture did not create the models, if only the automatic way was used. It could be argued that the algorithm is the one generating the models after

Table 8.7: Table of model vectors identified by the ARAVQ algorithm when applied to the state vector of figure 8.6. Association with states has been manually performed.

ARAVQ model	IHU S_x	IHU S_y	IHU M_l	IHU M_r	Manual model
1	0.9	0.11	0.49	0.96	a
2	0.7	0.1	0.58	0.91	d_1
3	0.37	0.1	0.74	0.74	d_2
4	0.17	0.11	0.82	0.58	d_3

processing the mixed and large quantity of data obtained from the IHU's. But, by comparing the results obtained manually, and the results obtained automatically, we have shown that the ARAVQ is not performing any extra processing, it simply performs automatically what had been manually performed.

8.4 Garbage collector behaviour state vector analysis

The example with the orbiting robot illustrated how the inner state vector works for a simple task in a simple robot, and how model vectors can be either manually or automatically identified. The state vector actually indicates an internal state for the robot, given by its current situation. Now, a more complex robot and task are selected to observe how the state vector behaves in such situation. The Khepera robot and the garbage collector problem are selected, as described and evolved in section 5.1, and the best of the DAIR controllers evolved in chapter 5 is used for the analysis.

8.4.1 The garbage collector state vector

The state vector for this robot is composed of the following components, related to the outputs of each IHU that compose the DAIR controller,

$$\begin{aligned} state\,vector(t) = \quad & (O_{S_A}(t), O_{S_B}(t), O_{S_C}(t), O_{S_D}(t), O_{S_E}(t), O_{S_F}(t), O_{S_G}(t), \\ & O_{M_l}(t), O_{M_r}(t), O_{P_t}(t), O_{P_r}(t)) \end{aligned}$$

where $O_{S_i}(t)$ is the output of the IHU associated to the sensor S_i for $i = A \ldots G$, $O_{M_j}(t)$ are the outputs of the IHU's for the motors, and $O_{P_k}(t)$ are the outputs of IHU procedures *take*, $k = t$, and *release*, $k = r$.

The problem to be faced when studying this state vector is that it contains a large number of components. Furthermore, when the robot detects something, a quick change between states is produced, which makes it difficult to observe a steady state in interesting situations like categorizing something detected as a stick. Each of the situations must be separately studied in a similar, but more complex manner than for the contour-following example.

8.4.2 Identified model vectors

In order to identify whether the robot evolved any set of stable well-defined states, some experiments were performed. The experiments consisted of allocating the robot in a special situation, and then to activate the obtained garbage collector controller using the tactical modularization. The values given by the sensor IHU modules were then measured. The situations selected were the following:

- the robot is moving in a free space with no obstacles around.

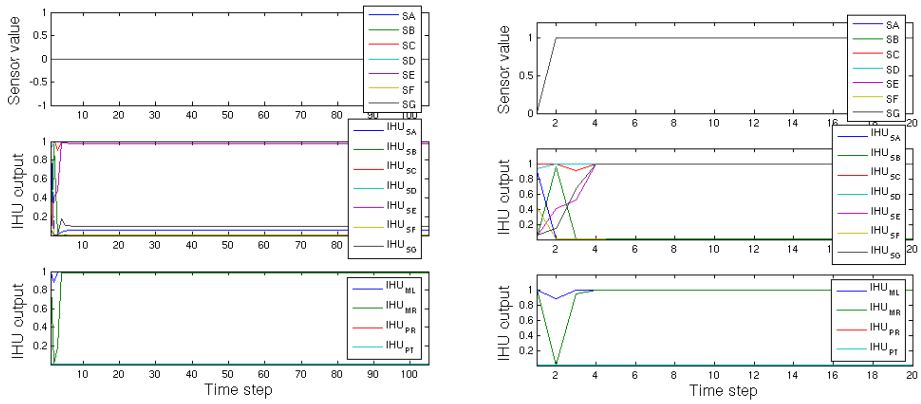


Figure 8.7: State vector for movement in free space, for the robot not carrying a stick (left) and for the robot carrying a stick (right). Both motor IHU's (IHU_M_l and IHU_M_r) maintain the same output value that makes the robot move forward until some object is detected.

- the robot detects a stick with different sensors.
- the robot detects a wall with different sensors.

All three different situations were tested with and without carrying a stick.

8.4.2.1 Moving around in a free space with no obstacles around

In this case the vector state remains in a steady state, moving both motors at the same speed. This behaviour sooner or later results in the robot sensing an obstacle, either a wall or a stick. Until this happens, the robot keeps on moving forward. The same behaviour is observed whether the robot is carrying a stick or not. The vector state is shown in figure 8.7 for several time steps. It is observed that the state vector remains constant in both cases, whether carrying or not carrying a stick. There are no oscillations of any type in the model vector, and both models are almost the same except for the fact that the IHU_S_G changes its value when carrying a stick.

The first model vector can be defined as follows:

Situation 1 Free space. This category is obtained when the robot detects nothing with its A to F sensors. The robot is put in the middle of the arena and no obstacles are beside it. After an initial transient time, the robot starts moving forward, assuming a stable state where the values of the IHU's outputs do not change at all. This state makes the robot advance forward. This behaviour ensures that the robot will eventually detect either the wall or a stick. The model vector values for this situation

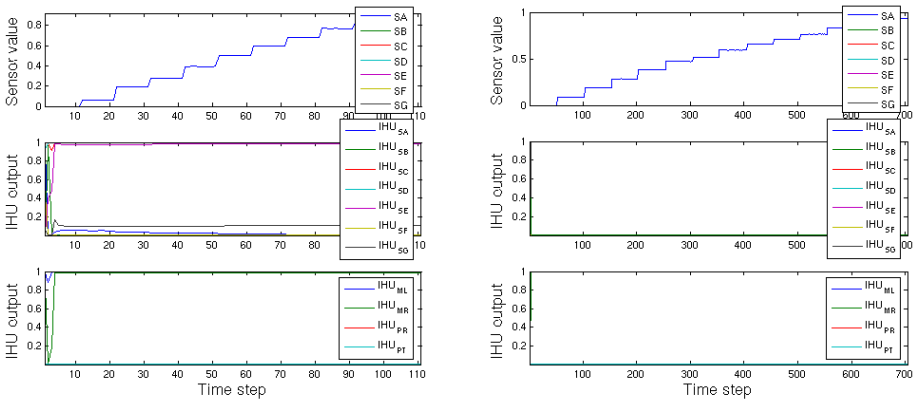


Figure 8.8: State vector of the robot when detects a stick with sensor A at different distances. The left figure is without carrying a stick, the right figure of the robot carrying a stick.

vary slightly in the IHU output, depending whether the robot is carrying the stick or not. The model vector values for this state are

- Without stick $(\textit{State-1-a})$ $\textit{state vector}(t) =$

$$(.05, 0, .99, .99, .97, 0, .10, .99, .98, 0, 0)$$

- With stick $(\textit{State-1-b})$ $\textit{state vector}(t) =$

$$(0, 0, .99, .99, .99, 0, .99, .99, .99, 0, 0)$$

8.4.2.2 Robot detects a stick with different sensors

This situation is produced when the robot is moving around the free space, and at some time, it detects a stick with one of its sensors. This situation is reproduced by placing the robot in front of a stick at different distances, and only in front of a single sensor. The same situation is repeated for each of the robot sensors, and state vector variations are observed under those circumstances, looking for the emergence of steady states as in the case of the contour-following behaviour.

Robot detects a stick with sensor A. In this case, the state vector acquires a constant value for all the range of measured distances by sensor A. In fact, it does not discriminate against the state whether it carries a stick or not. Furthermore, the vector is the same, in practical terms, as in the case of the robot in a free space. The model vector can be defined as follows,

Situation 2 The model vector values for this state are:

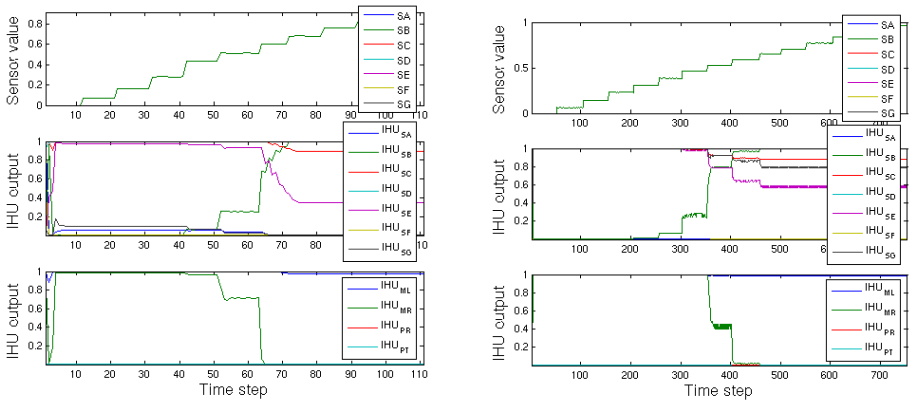


Figure 8.9: State vector of the robot when detecting a stick with sensor B at different distances. The left figure is without carrying a stick, the right figure is the robot carrying a stick.

- Without stick (*State-2-a*) $state\,vector(t) =$

$$(.01, 0, .99, .99, .97, 0, .09, .99, .98, 0, 0)$$

- With stick (*State-2-b*) $state\,vector(t) =$

$$(0, 0, .99, .99, .99, 0, .99, .99, .99, 0, 0)$$

By comparing the model vectors obtained for state-1 and for state-2, it is observed that both represent exactly the same state. This means that the robot is treating both situations as the same, either when is not detecting something or when is detecting something with the sensor A. For the robot, both situations provide the same information related to the task at hand, that is, they have the same meaning, from the robot's point of view.

The robot detects a stick with sensor B. In this case the robot detects the stick with sensor B. When the stick is detected closer than a certain threshold distance provided by the associated IHU, $SB = 0.45$ in this case, then the robot avoids the stick, - either while it is carrying a stick or while it is not, by reducing the speed of the right motor. In the first instance, the robot just reduces the speed a little, enough to avoid the stick on time. But, if the distance to the stick is smaller than a second IHU-emerged threshold, $SB = 0.6$, then a complete emergency turn is activated by motor right going backwards. The same situation can be observed in the case of the robot carrying a stick. Three different model vectors exist for this behaviour.

Situation 3 The model vector values for this state are:

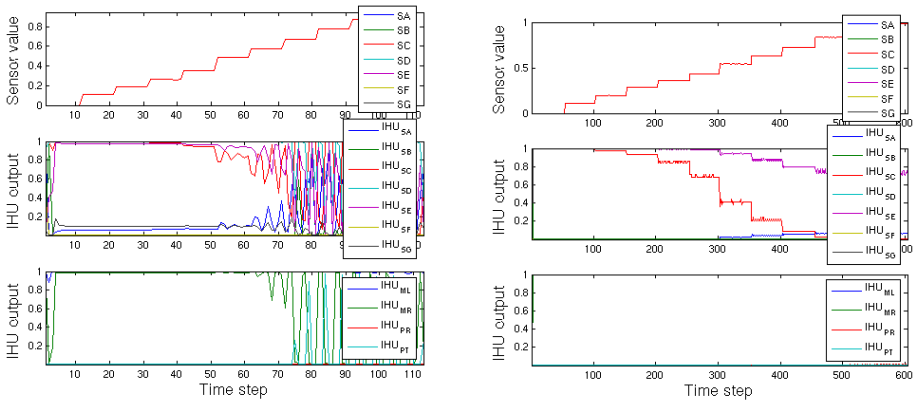


Figure 8.10: State vector of the robot when detects a stick with sensor C at different distances. The figure on the left is the case without carrying a stick, on the right for the robot carrying a stick.

- Without stick (*State-3-a-{1,2,3}*) *state vector*(t) =

$$\begin{array}{l}(.05, 0, .99, .99, .97, 0, .10, .99, .98, 0, 0)\ ,\ SB < .45\\ (.03, p1, p2, 1, p3, 0, .02, 1, .71, 0, 0))\ ,\ SB \in [.45, .6]\\ (0, .99, .89, 1, .34, 0, 0, .97, 0, 0, 0)\ ,\ SB > .6\end{array}$$

- With stick (*State-3-b-{1,2,3}*) *state vector*(t) =

$$\begin{array}{l}(0, 0, .99, 1, .99, 0, .99, .99, .99, 0, 0)\ ,\ SB < .45\\ (0, p1, p2, 1, p3, 0, p4, 1, .45, 0, 0)\ ,\ SB \in [.45, .6]\\ (0, .99, .88, 1, .58, 0, .79, .98, 0, 0, 0)\ ,\ SB > .6\end{array}$$

where p_k, $k = 1, \ldots 4$ are proportional values to the sensed value SB.

Robot detects a stick with sensor C. Let's first describe the situation experienced by the robot when it is carrying a stick (figure 8.10-right). In this case, the robot recognizes the different distances at which the stick is detected, by changing the values of the IHU-S_C and IHU-S_E. Their value decreases as the sensor value increases. However, the strategy of the motors does not change at any point, that is, the robot turns at maximum speed at any distance detected by the sensor, trying to avoid the stick in front of it.

In the case that the robot is not carrying a stick (figure 8.10-left), the state remains stable up to a certain detection threshold in sensor C. If the distance detected corresponds to a sensor value below 0.45, the robot will act as if it was in a free space. Once the sensor detects the stick closer than that distance, it will activate the *take* procedure. At the same time, it will reduce the speed of the right motor. This is due to the fact that the robot was trained for the replacement of a stick in front of it once one was picked up.

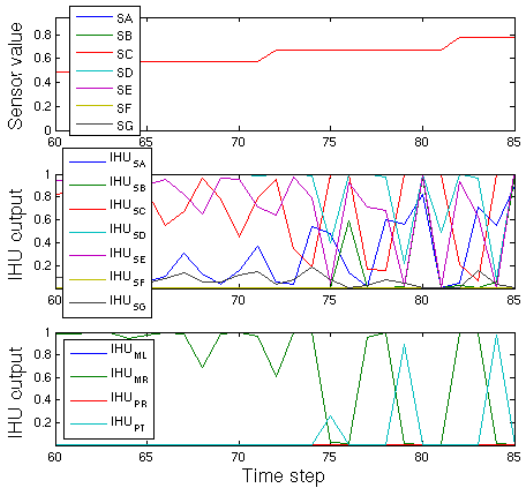


Figure 8.11: Close-up of the state vector of the robot when detects a stick with sensor C at a distance closer than 0.45. The internal state becomes oscillatory.

The state of the robot is expressed as an oscillatory pattern, as is shown in figure 8.11. Those IHU's that oscillate all perform the oscillation at the same frequency. This makes sense, since all the IHU's are coupled. Another observation is that the amplitude of the oscillation varies with the distance to the object, that is, when the distance to the object decreases, the amplitude of the oscillation increases. This amplitude reaches a maximum value when the value sensed by sensor S_C reaches the 0.6 value. After that value, the oscillation maintains its amplitude for any other sensed value above that threshold.

The IHU's involved in the oscillation are: IHU-S_A, IHU-S_B, IHU-S_C, IHU-S_D, IHU-S_E, IHU-S_G, IHU-M_r, IHU-P_t. Appendix F shows the figures of the time evolution of each IHU and their spectral components calculated from the Fast Fourrier Transform of their time series. The figures show how signals oscillate at 2 Hz, given that each time step is of 100 ms of duration.

Situation 4 Based on the oscillation frequency of the different signals the model vector values for this situation are:

- Without stick (*State-4-a-{1,2}*) $state\,vector(t) =$

$$(.05, 0, .99, .99, .97, 0, .10, .99, .98, 0, 0)\ ,\ SC \leq .45$$
$$(p_{Sa}s(0), p_{Sb}s(-.2378), p_{Sc}s(-.0858), p_{Sd}s(2.08), p_{Se}s(1.71),$$
$$0, p_{Sg}s(2.21), .94, p_{Mr}s(2.98), p_{Pt}s(-1.37), 0)\ ,\ SC > .45$$

with $s(n) = \sin(4\pi + n)$.

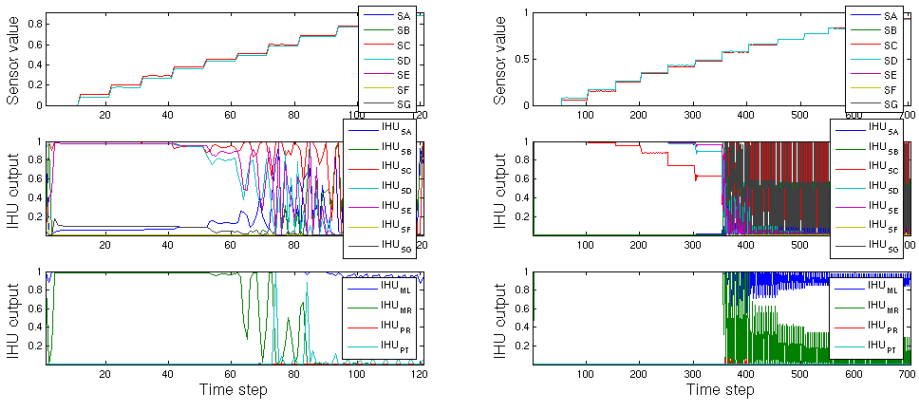


Figure 8.12: State vector of the robot when detects a stick with sensors C and D, at different distances. When not carrying a stick (left) and when carrying it (right).

- With stick (*State-4-b-{1,2}*) *state vector*(t) =

$$(0, 0, .95, .99, .99, 0, .99, .99, .99, 0, 0)\ ,\ SC \leq .1$$
$$(01, 0, p1, .99, p2, 0, .99, .99, .99, 0, 0)\ ,\ SC > .1$$

  where $p1$, $p2$, $p_{Sa}, p_{Sb}, p_{Sc}, p_{Sd}, p_{Se}, p_{Sg}, p_{Mr}, p_{Pt}$ take proportional values to the sensor distances detected.

The situation State-4 contains a periodic signal of a certain frequency. This state will be defined as a *non-tonic state*, that is, a state which is bounded, but it is not steady. On the other hand, States 1 to 3 will be defined as *tonic states*. In the case of the contour-following experiment, all the states are tonic.

The robot detects a stick with both sensors C and D. When the robot is not carrying a stick, the state vector presents three different model vectors; two of them being non-tonic states. The three different state vectors depend on the distance detected by the sensors. When the robot distance detected is below 0.45, according to the IHU interpretation, the state vector is the same as the robot being in free space. When the sensor value is above 0.45 the first non-tonic state appears, with the robot capturing the stick (activation of the *take procedure*, and reduction of the right motor velocity). On the other hand, when the distance detected is above 0.6 another non-tonic state appears, where a simple avoidance behaviour is activated by reducing the speed of the right wheel.

Situation 5 The model vector values are:

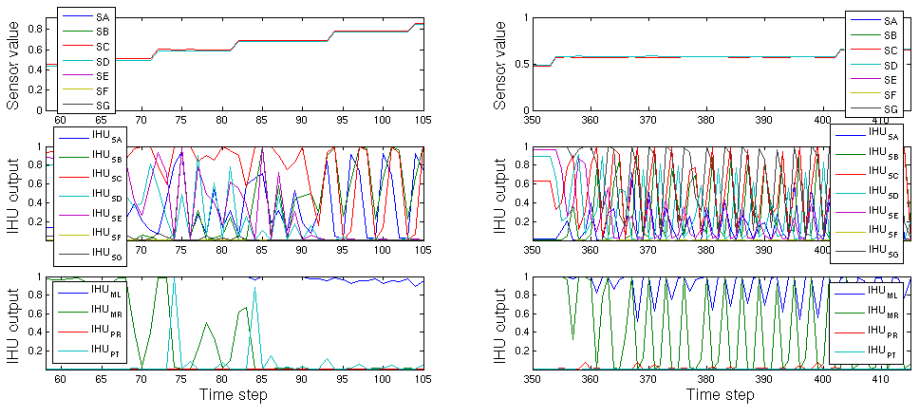


Figure 8.13: Close-up of the state vector of the robot when it detects a stick with sensors C and D, at different distances. For the case without carrying a stick (left), and for the robot carrying a stick (right).

- Without stick (*State-5-a-{1,2,3}*) *state vector*(t) =

  $(.05, 0, .99, .99, .97, 0, .10, .99, .98, 0, 0)$, $SC, SD < .45$
  $(p_{Sa}s(0), 0, .9, p_{Sd}s(2.158), p_{Se}s(2.632), 0, 0, .99, p_{Mr}s(3.082),$
  $p_{Pt}s(-0.995), 0)$, $SC, SD \in [.45, .6]$
  $(p_{Sa}s(0), p_{Sb}s(1.189), p_{Sc}s(1.4468), 0, 0, 0, 0, .99, 0, .05, 0)$,
  $SC, SD > .6$

  with $s(n) = \sin(4\pi + n)$ for *State-5-a-2* and $s(n) = \sin(5\pi + n)$ for *State-5-a-3*.

- With stick (*State-5-b-{1,2}*) *state vector*(t) =

  $(0, 0, .95, .99, .99, 0, .99, .99, .99, 0, 0)$, $SB < .5$
  $(p_{Sa}s(0), p_{Sb}s(1.99), p_{Sc}s(-.942), p_{Sd}s(1.04), p_{Se}s(1.93), 0,$
  $p_{Sg}s(-2.07), p_{Ml}s(1.15), p_{Mr}s(-1.76), 0, 0)$, $SB \geq .5$

  with $s(n) = \sin(6.5\pi + n)$.

The robot detects a stick with sensor D. As in the previous case of sensors C and D, when the robot does not carry a stick two non-tonic states are observed: when the distance detected is above 0.45, the robot turns to the right by reducing its right wheel speed (this behaviour will place the robot on the same spot as the stick's previous location, as detected by sensors C and D); and when the distance detected is above 0.6, the robot will just activate the take procedure. In the case of the robot carrying a stick, the behaviour of the robot remains the same for all the distances, but its internal state does not. When above a threshold of 0.45, the inner state of the robot starts to progressively change.

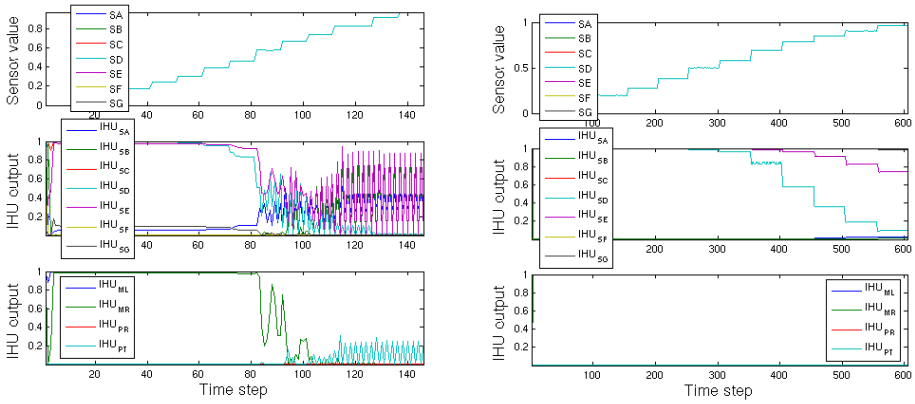


Figure 8.14: State vector of the robot when it detects a stick with sensor D at different distances. The figure on the left is when not carrying a stick, and figure on the right when carrying a stick.

It is interesting to observe how, in the case of the robot carrying a stick the internal state changes, the IHU output of the sensors changes but not of the actuators, even if no action is taken. This implies that not all internal states have an associated action, but just pure internal states. In fact, any human meaning to that situation can be assigned, but the robot has found it worthwhile to generate such state specification.

Situation 6 The model vector values for this state are:

- Without stick (*State-6-a-{1,2,3}*) *state vector*$(t) =$

  $(.05, 0, .99, .99, .97, 0, .10, .99, .98, 0, 0), SD < .45$
  $(p_{Sa}s(0), 0, .99, p_{Sd}s(1.68), p_{Se}s(-2.16), 0, 0, 1, p_{Mr}\bar{s}(0), .01, 0)$,
  $SD \in [.45, .6]$
  $(p_{Sa}s(0), .05, 1, p_{Sd}s(1.68), p_{Se}s(-2.16), 0, 0, 1, 0, 0,$
  $p_{Pt}s(-1.267))$, $SD > .6$

  with $s(n) = \sin(10\pi + n)$, $\bar{s}(n) = \sin(2\pi + n)$ for *State-6-a-2* and $s(n) = \sin(6.5\pi + n)$ for *State-6-a-3*.

- (b) With stick (*State-6-b-{1,2}*) *state vector*$(t) =$

  $(0, 0, .99, .99, .97, 0, 1, .99, .99, 0, 0)$, $SD < .45$
  $(0, 0, .99, p1, p2, 0, 1, 1, 1, 0, 0)$, $SD \geq .45$

  with $p1$ and $p2$ being proportional values to SD.

Robot detects a stick with sensor E. In both situations, - whether carrying or not carrying a stick, there are two possible states, both of them tonic:

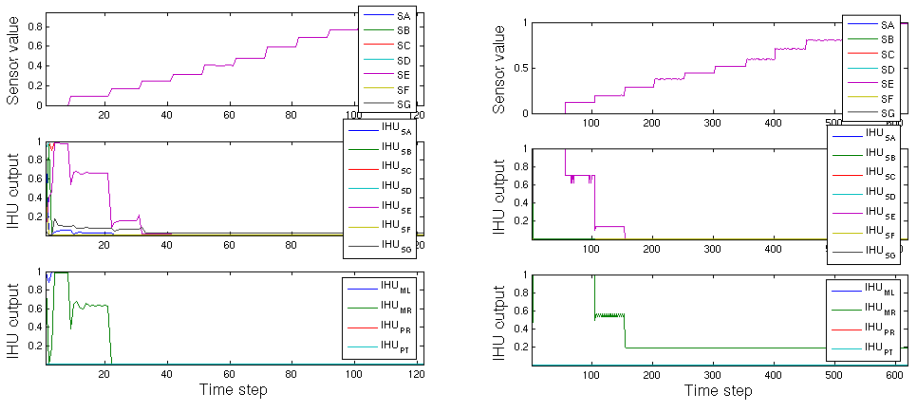


Figure 8.15: State vector of the robot when detects a stick with sensor E at different distances. The left figure is when not carrying a stick, the right figure of the robot when carrying a stick.

when the robot is far from the stick and decides to turn to it smoothly, at a speed proportional to the distance detected; and when the robot just rotates at a maximal speed.

Situation 7 The model vector values for this situation are:

- Without stick (*State-7-a-*{*1,2*}) $state\,vector(t) =$

$$(.02, 0, .99, .99, p1, 0, .10, .99, p2, 0, 0)\ ,\ SE < .2$$
$$(0, 0, .99, .99, 0, 0, .02, .99, 0, 0, 0)\ ,\ SE \geq .2$$

  with $p1$ and $p2$ proportional values to SE for *State-7-a-1*.

- With stick (*State-7-b-*{*1,2*}) $state\,vector(t) =$

$$(0, 0, .99, .99, p1, 0, 1, .99, p2, 0, 0)\ ,\ SE < .2$$
$$(0, 0, .99, .99, 0, 0, 1, 1, .19, 0, 0)\ ,\ SE \geq .2$$

The robot detects a stick with sensor F. Basically, the controller pays no attention to what is detected by sensor F.

Situation 8 Model vector values are:

- Without stick (*State-8-a*) $state\,vector(t) =$

$$(0, 0, .99, .99, .99, 0, .99, .99, 1, 0, 0)$$

- With stick (*State-8-b*) $state\,vector(t) =$

$$(.05, 0, 1, 1, .97, 0, .1, 1, .98, 0, 0)$$

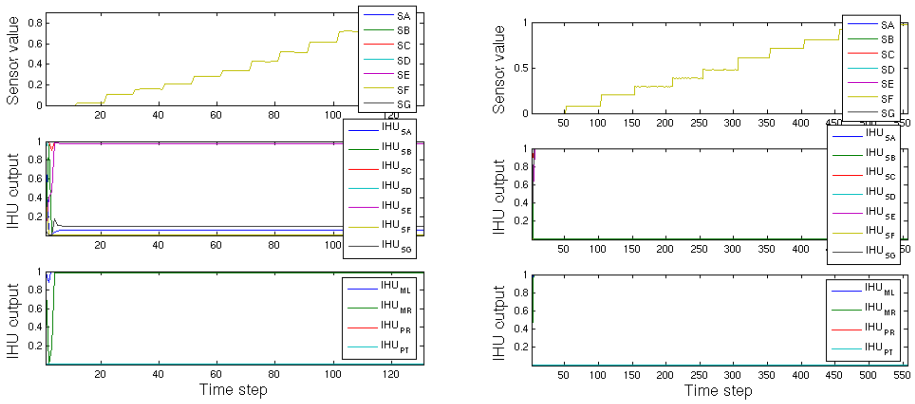


Figure 8.16: State vector of the robot when detects a stick with sensor F at different distances. The left figure is the case without carrying a stick, the right figure for the robot carrying a stick.

8.4.2.3 The robot in front of a wall at different angles

These situations are produced when the robot is moving around the free space, and at some stage it detects a wall with one or more of its sensors. The same experimental analysis as in the previous section is performed. The robot is placed in front of a wall with one sensor at a time pointing towards it. The robot is then manually moved towards the wall and the resulting state vector is plotted.

The robot detects a wall with sensor A. For the case of the robot not carrying a stick, three different situations can be distinguished: when the robot does nothing, when the robot produces a turn proportional to the values sensed, and when the robot turns at full speed. For the case of the robot carrying a stick, the same three situations can be observed, except that the output of the motors is on-off.

Situation 9 The model vector values for this state are:

- Without stick ($State\text{-}9\text{-}a\text{-}\{1,2,3\}$) $state\,vector(t) =$

$$(.01, 0, .99, .99, .98, 0, .1, .99, .98, 0, 0)\ ,\ SA < .76,\ SB < .49$$
$$(0, .99, .89, .99, .35, 0, 0, .97, 0, 0, 0)\ ,\ SA > .87,\ SB > .65$$
$$(0, p1, p2, .99, p3, 0, .03, .99, p4, 0, 0)\ ,\ \text{otherwise}$$

- With stick ($State\text{-}9\text{-}b\text{-}\{1,2,3\}$) $state\,vector(t) =$

$$(0, 0, .99, .99, .99, 0, .99, .99, .99, 0, 0)\ ,\ SA < .75,\ SB < .47$$
$$(0, .99, .87, .99, .57, 0, .79, .97, 0, 0, 0)\ ,\ SA > .94,\ SB > .75$$
$$(0, p1, .98, .99, .97, 0, .99, .99, .99, 0, 0)\ ,\ \text{otherwise}$$

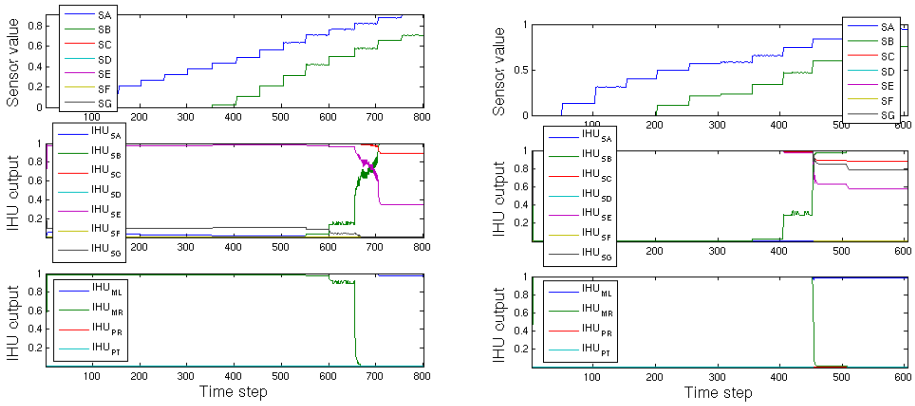


Figure 8.17: State vector of the robot when it detects a wall with sensor A at different distances. The left figure is the case without carrying a stick, the right figure for the robot carrying a stick. Both figures show that at a given distance of the wall, the robot detects it with two sensors S_a and S_B, due to their proximity.

The robot detects a wall with sensor B. Interactions between the different IHU's make very difficult to analyze those states. Furthermore, some of the states are really quick and short transitions in time. However, we can provide a simple interpretation. In both cases, carrying and not carrying a stick, the robot continues straight forward, with both wheels travelling at the same speed, until the robot reaches a distance close to the wall. At that point, the robot just tries to avoid the wall in both cases, by decreasing the speed of its right wheel.

State 10 It can be described as follows:

- Without stick (*State-10-a*) $state\,vector(t) =$

$$(.04, 0, .99, .99, .97, 0, .10, .99, .98, 0, 0)\,, \text{ up to step } 500$$
$$(.06, .99, p1, p2, 0, 0, 0, .72, 0, 0, 0)\,, \text{ from step } 500$$

  wtih $p1$ and $p2$ varying.

- With stick (*State-10-b*) $state\,vector(t) =$

$$(0, .0, .99, .99, .99, 0, .99, .99, .99, 0, 0)\,, \text{ up to step } 265$$
$$(.1, 1, .01, 0, 0, 0, .05, .69, 0, 0, .01)\,, \text{ from step } 265$$

The robot detects a wall with sensors C and D This is another non-tonic state with complex relations between different IHU's. The situation is described as:

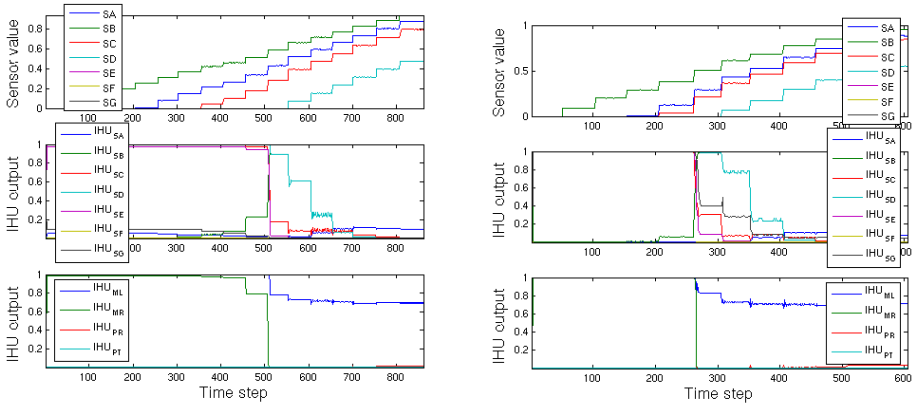


Figure 8.18: State vector of the robot when detects a wall with sensor B at different distances. The left figure is of the robot not carrying a stick, the right figure of the robot carrying a stick.

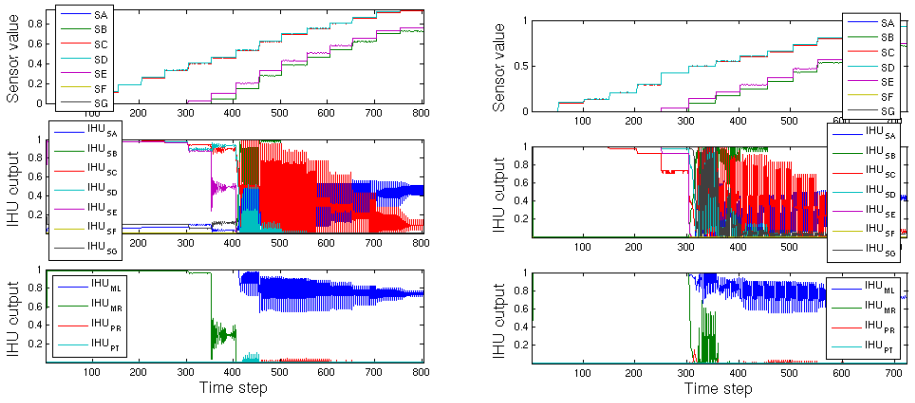


Figure 8.19: State vector of the robot when it detects a wall with sensors C and D at different distances. The left figure is of the robot not carrying a stick, the right figure of the robot carrying a stick.

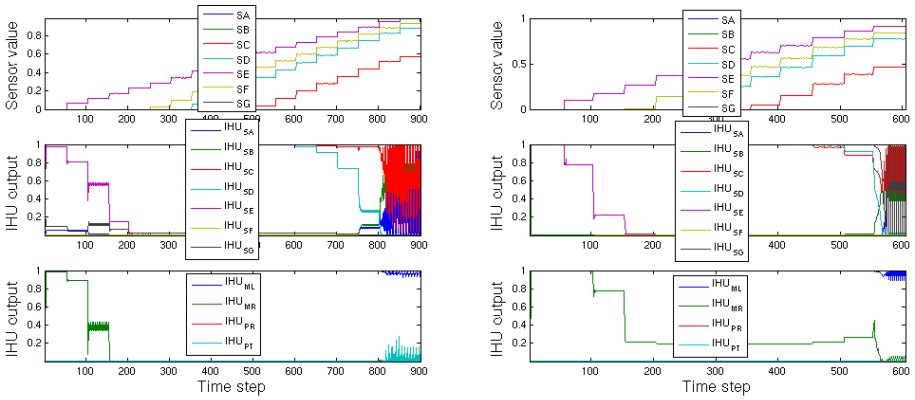


Figure 8.20: State vector of the robot when it detects a wall with sensor E at different distances. The left figure is of the robot not carrying a stick, the right figure of the robot carrying a stick.

Situation 11 The model vector values for this state are:

- Without stick ($State\text{-}11\text{-}a\text{-}\{1,2,3\}$) $state\,vector(t) =$

  $(.05, 0, 1, 1, .97, 0, .1, 1, .98, 0, 0)$
  $(.1, 0, .9, .9, .5, 0, 0, .99, .3, 0, 0)$
  $(p_{Sa}s(0), p_{Sb}s(2.57), p_{Sc}s(1.46), p_{Sd}s(0.395), .97, 0, .05, p_{Ml}s(0.44),$
  $p_{Mr}s(-1.06), 0, p_{Pr}s(-1.79))$

  with $s(n) = \sin(5\pi + n)$ for *State-10-a-3*.

- With stick ($State\text{-}11\text{-}b\text{-}\{1,2,3\}$) $state\,vector(t) =$

  $(0, 0, .99, .99, .99, 0, .99, .99, .99, 0, 0)$
  $(p_{Sa}s(0), p_{Sb}s(1.72), p_{Sc}s(1.13), p_{Sd}s(3.12), 0, 0, p_{Sg}s(0.41),$
  $p_{Ml}s(-1.14), p_{Mr}s(-0.14), 0, 0)$
  $(p_{Sa}s(0), p_{Sb}s(1.99), p_{Sc}s(0.94), p_{Sd}s(1.92), 0, 0, p_{Sg}s(-2.501),$
  $p_{Ml}s(0.126), 0, 0, 0)$

  with $s(n) = \sin(6.5\pi + n)$ for *State-11-b-2* and $s(n) = \sin(5\pi + n)$ for *State-11-b-3*.

The robot detects a wall with sensor E. The situation can be defined as follows:

Situation 12 It takes the following values for the model vectors,

- Without stick ($State\text{-}12\text{-}a\text{-}\{1,2,3\}$) $state\,vector(t) =$

  $(0, 0, .99, .99, .99, 0, .10, .99, .99, 0, 0)$
  $(0, .1, .9, .9, 0, 0, 0, 1, 0, 0, 0)$
  $(p_{Sa}s(0), p_{Sb}s(.675), p_{Sc}s(1.16), 0, 0, 0, 0, 1, 0, 0, 0)$

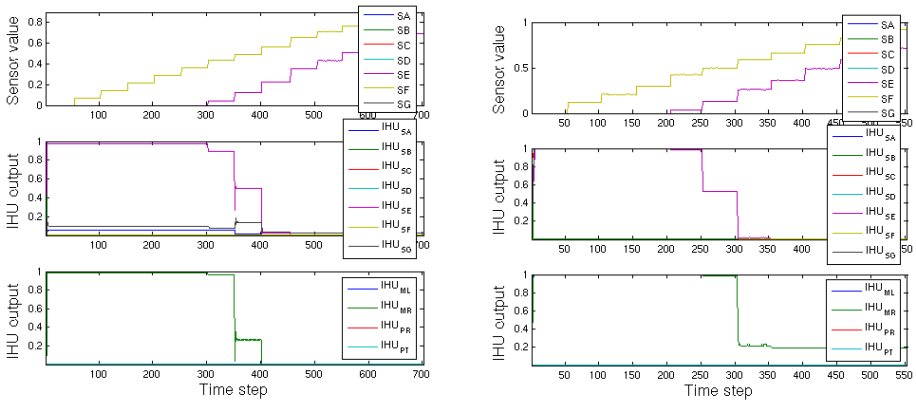


Figure 8.21: The state vector of the robot when it detects a wall with sensor F at different distances. The left figure is the case without carrying a stick, the right figure for the robot carrying a stick.

with $s(n) = \sin(5.7\pi + n)$ for *State-12-a-3*.

- Carrying a stick (*State-12-b-{1,2,3}*) *state vector*$(t) =$

  $(0, 0, .99, .99, .99, 0, .10, .99, .99, 0, 0)$
  $(0, .1, .9, .9, 0, 0, 1, 1, .2, 0, 0)$
  $(p_{Sa}s(0), p_{Sb}s(-5.53), p_{Sc}s(-.67), 0.1, 0, 0, p_{Sg}s(-1.02), 1, 0, 0, 0)$

  with $s(n) = \sin(6.8\pi + n)$ for *State-12-b-3*.

The robot detects a wall with sensor F When the robot experiences this situation, its behaviour is more or less the same; either it is carrying or it isn't carrying a stick. In both cases, the robot does not pay any attention to the wall until sensor E also starts to detect the wall. When this happens, the robot reduces the speed of the right motor in order to turn to the right. All the states observed are tonic states.

Situation 13 The model vector values for this state are:

- Without stick (*State-13-a-{1,2}*) *state vector*$(t) =$

  $(.05, 0, .99, .99, .97, 0, .10, .99, .98, 0, 0)\,,\ SE < .1$
  $(.05, 0, .99, .99, p1, 0, .10, .99, p2, 0, 0)\,,\ SE > .1$

  with $p1$ and $p2$ proportional values to the S_E sensor up to the minimum value of 0.0.

- With stick (*State-13-b-{1,2}*) *state vector*$(t) =$

  $(.05, 0, .99, .99, .97, 0, .10, .99, .98, 0, 0)\,,\ SE < .1$
  $(.05, 0, .99, .99, p1, 0, .10, .99, p2, 0, 0)\,,\ SE > .1$

α	δ	ε	n
0.03	0.2	0.4	3

Table 8.8: ARAVQ parameters used for the extraction of states in the garbage collector behaviour.

with $p1$ and $p2$ proportional values to the S_E sensor up to the minimum value of 0.0 for $p1$ and 0.2 for $p2$.

8.4.3 Automatic extraction of states

In the case of the garbage collector problem, the automatic extraction of states can be applied up to a certain point. The ARAVQ method employed is not able to detect oscillatory states since the algorithm is based on a moving average. However, it is observed that during a normal execution of the garbage collector controller, oscillatory states are never produced, since the whole controlling situation is performed in a continuous way. Instead of oscillatory states, a *dance* of IHU values is obtained, modifying the situation the robot will experience in the next step, avoiding the robot being trapped in an oscillatory state.

However, even if no oscillations appear, in most of the situations found by the garbage controller, the controller state only lasts for a single time step, preventing this fact the detection of those states by ARAVQ. Then, neither is it possible to observe the state in a steady manner, as happened with the contour-following, nor with an ARAVQ analysis, since the duration of the states is very short. An application of the ARAVQ algorithm to a complete run of the garbage collector provides, of course, a set of detected states, but just a few of them are identified because short duration in time for other ones, in fact, a single time step. In table 8.9 the situations identified for the ARAVQ algorithm when applied to a typical run of the garbage collector can be observed [3], using parameters displayed in table 8.8.

More than the results obtained, what is interesting here, is that the current state of the robot can be identified by looking at the internal state vector, even if the state only lasts for a single time step. A better way of automatically extracting the states is work for the future.

8.5 Aibo behaviours state vector analysis

The vector state for any Aibo controller evolved in previous chapters contains too many components to analyze manually, as has been completed in sections 8.3 and 8.4 for the contour-following controller and the garbage collector. Furthermore, it is not possible to put the Aibo robot in a static defined situation to observe the evolution of its state. The only way to analyze the state vectors and discover its model vectors is by using the automatic approach while the

[3] Video of the run used is available at www.ouroboros.org/thesis

Table 8.9: List of model vectors discovered by the ARAVQ algorithm when parameters in table 8.8 are used.

$IHU-$	S_A	S_B	S_C	S_D	S_E	S_F	S_G	M_L	M_R	P_T	P_R
1	0	0	1	1	1	0	0	1	1	0	0
2	0	0	1	1	1	0	1	1	1	0	0
3	.2	1	.45	.2	0	0	0	.84	0	0	0
4	0	.67	1	1	0	0	0	1	0	0	0
5	0	0	1	1	.3	0	1	1	.55	0	0
6	0	1	.6	1	0	0	0	1	0	0	0
7	0	.6	1	1	.67	0	0	1	0	0	0
8	0	1	1	1	.8	0	1	1	.35	0	0
9	0	0	.4	.8	0	0	1	1	.4	0	0
10	.2	0	.5	1	.65	0	0	1	.7	0	0

Table 8.10: ARAVQ parameters used, and results obtained for the extraction of states in the Aibo standup behaviour. MA stands for moving average and Mv for Model vector. d stands for deviation.

α	δ	ε	n	# models	MA d.	Mv d.
0.03	0.02	0.6	10	8	0.341	0.35
0.03	0.2	0.6	10	2	0.341	0.356
0.03	0.02	0.06	10	0	0.341	0.08

robot is performing the required task. Just to have an idea of what the states would look like, the ARAVQ algorithm was applied to extract some of the possible states that the controller is generating[4]. The following sections show the results obtained when applied to two examples: the Aibo stand up and the Aibo walking behaviours.

8.5.1 Aibo stand up

The final parameters used in the ARAVQ algorithm are displayed in table 8.10. Depending on the values of the parameters the number of different vectors found ranges from 8 to 1 (with the parameters used). This implies a good reduction of the sensory state but only from a static analysis of the evolution of the state vector. More complex analysis may be required to really obtain the relevant states for the task, like happenned in the garbage collector problem.

8.5.2 Aibo walking

[4]Future work will be to implement a better system that automatically extracts all the states created by a giving controller.

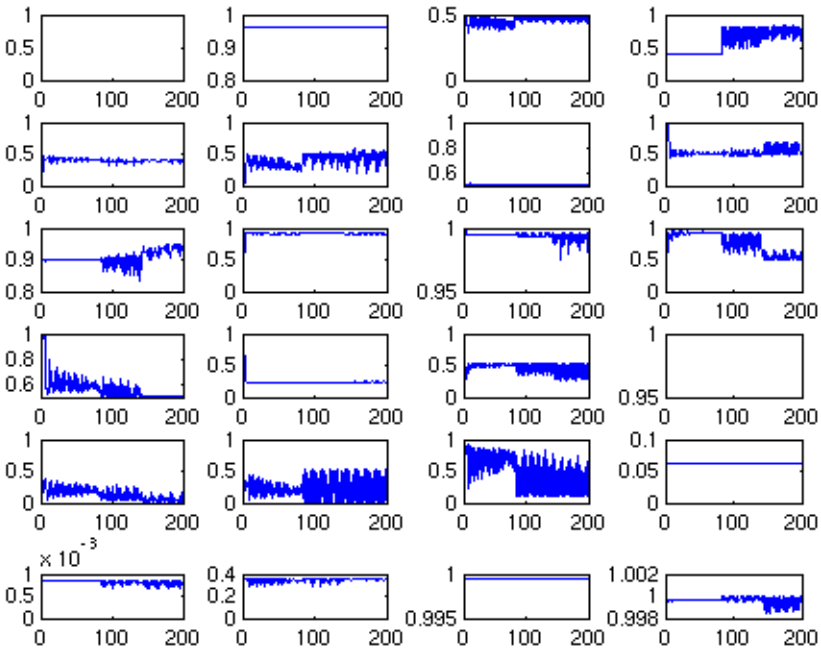


Figure 8.22: Aibo stand up behaviour state vector. Each plot shows, from top to bottom, and from left to right, the IHU output of sensors.

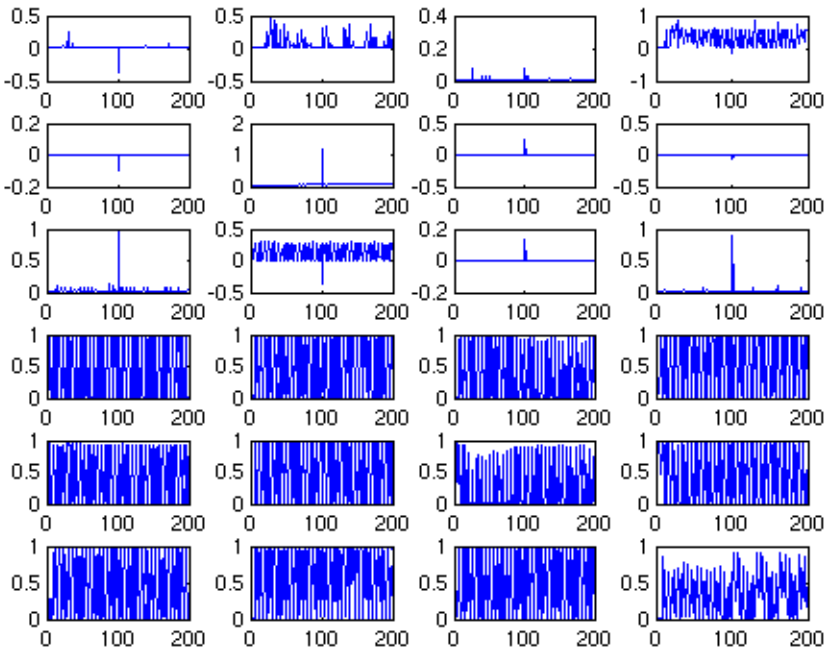


Figure 8.23: Aibo walking behaviour state vector.

Table 8.11: ARAVQ parameters used for the extraction of states in the Aibo walking behaviour

α	δ	ε	n	# models	MA d.	Mv d.
0.03	0.002	0.6	2	6	0.8	1.765
0.03	0.002	0.6	10	0	1.198	0.6
0.03	0.02	0.06	2	0	0.8	0.8

The final parameters used in the ARAVQ algorithm are displayed in table 8.11. The same situation as in the Aibo standup task is observed for this case. A more complex analysis is required to really observe the model vector which are relevant for the task.

8.6 Discussion

From the results obtained analyzing the *contour-following* behaviour and the *garbage collector* behaviour, it can be observed that DAIR controllers for each behaviour produce similar output patterns in similar situations. The sensor IHU's provided the same output values to different sensor values which corresponded to the same *conceptual* situation. Therefore, the sensor IHU's were classifying a number of different sensory states all into the same conceptual category or meaning. The different categories or meanings can be accessed by using what we call the *state vector* of the robot at a given time step. The state vector is formed by the concatenation of the output values of the sensors IHU's at each time step.

The state vector identifies the situation of the robot for each time step. Basically, it can be seen as a categorization of its current situation, or as an internal modelling of the outside world that the robot is experiencing at that particular moment. This internal representation at the IHU level contains the meaning of the situation, and that meaning is attached to the present sensor activity pattern. Changes in the values of the sensors did not change the state vector, unless a change in the situation of the robot, relevant for the task to solve, was produced. Changes from one state to another one are not instantaneous and they involve a transient time where the IHU's exchange information and finally adopt the new state.

The internal representations that map the sensory stimulation to the category actually being experienced are automatically created by the evolutionary process while interacting with the environment. Hence, the meanings are grounded on the robot experiences. This means that the actual states identified by the robot have to signify meaning for the robot. However, this meaning does not have to correspond to a human meaning, but rather a meaningful state for the robot about the task to be solved.

In fact, we think that this is exactly what actually happens when in the case of the garbage collector, changes in the sensor values generate changes in the

sensor IHU's values, but the actuator IHU's mantain the same output (action). This can be seen as an internal meaning (or state) that we cannot correspond to a human meaning.

For the garbage collector problem, the robot identifies only a few possible states as required for the solution of the task at hand, allowing it to reduce the huge number of possible sensor inputs and robot states to a reduced number of relevant ones. A group of sensors values will always correspond to a unique single meaning or category. It represents a huge reduction from the high number of possible situations that raw sensed data provide. The internal states created by the system identify those states that have a real semantic value, and that value is grounded to the experiences of the robot.

A meaning can be attributed to each of those vectors, from an external human point of view. It should be clearly stated that those meanings have been assigned by us (humans) to the robot's different situations, and that the robot is not aware of it knowing those meanings. However, what the robot has done is the automatic creation of a categorization of situations, that is, the robot has emerged a system by which the current situation that it is experiencing is described by the state vector. The robot categorizes its current situation into a simple set of possible ones, and all the situations are categorized into one of these possible categories.

The advantage of the DAIR architecture against other modular and non-modular architectures is that the categorization emerged is directly accessible to an observer outside of the network-based controller, that is, the meanings are not internally coded in the network weights. This means that it is possible to directly access the robot's situation from a conceptual point of view simply by looking at the IHU sensor outputs. This type of direct access to the generated meanings may not be necessary in biologically intelligent systems, but scientists feel more comfortable when such differentiation is possible as it makes the whole process easier to understand. Furthermore, it may assist in maintaining the correspondence between syntax and semantics. This could be achieved by accessing the meanings created by a more deliberative superior layer, which would use them to (syntactically) *think* about its situation, in this way propagating the robot-acquired meanings to more syntactic processes, as a kind of reification engine [Gunderson and Gunderson, 2009].

From another point of view, the actuation of the architecture can be seen as an extractor of meaningful events which are relevant for the resolution of the task. The architecture is capable of converting a continuous flow of sensory data into a discrete number of meaningful situations. We will call these situations *events*. A new event is generated each time that the robot *thinks* the situation changes. And the situation changes when the robot itself *thinks* that the new sensory flow corresponds to something really different from previous situation; it in fact creates a categorization of experiences that are useful for the task at hand. This behaviour is similar to the ARAVQ event extractor algorithm [Linaker and Niklasson, 2000], with the difference that the ARAVQ extracts events from the information gathered by a robot that already knows how to solve the task. The introduced architecture, on the other hand, learns to extract

the events whilst simultaneously learning how to perform the task.

When categorizing, an agent situated in the real world has to be able to make distinctions between different types of objects and situations from the sensed values. This subject has been studied by others researchers [Nolfi, 1997, Pfeifer and Scheier, 1997]. They proposed sensorimotor coordination as the key to categorization, and argued that it is necessary to replace the information processing metaphor with a sensorimotor metaphor. This sensorimotor approach has been used in several studies like [Choe and Bhamidipati, 2004], where sensorimotor couplings assigned a meaning to the sensor state through sensory-invariance driven action, or [Philipona et al., 2003], where the external space of the robot was inferred from sensorimotor dependencies.

These observed states indicate that the DAIR architecture indeed uses the sensorimotor coordination metaphor to produce its categorization. The clearest example is the result obtained in *Situation 3* for the garbage collector problem, when the robot detects something but cannot identify what it is. This situation indicates that the robot is having *perceptual aliasing*. Its strategy to solve the situation is to move itself into a more convenient position which provides it with a more convenient sensor input which in turn permits it to better determine what is in front of it. This type of behaviour is what has been referred to as *active perception* [Nolfi and Marocco, 2002]

Another consequence of the results obtained here is that they may lead to the automatic generation of rules from neural networks.

8.7 Conclusions

An analysis of the inner workings of the DAIR architecture has been provided. The analysis has only been performed for reactive behaviours and situations (orbit behaviour and garbage collector). We have shown how a DAIR-based controller is able to create a meaningful internal representation of the robot's current situation directly grounded on its sensorimotor system. In simple cases a manual detection of the robot meanings has been performed, and even an automatic way based on ARAVQ has been used. For more complex cases, like the walking for Aibo, other more complex analysis techniques may be required due to the fact that they include internal states within their IHU modules, hence complexifying the analysis of the state vector. Proposed solutions are: to use neural nets with recurrent connections to discern the states, or to use dynamical systems (in a similar way as has been done in [Montebelli et al., 2008]).

What is important here is not whether we have identified a set of stable states or whether we have been able to assign a meaning to them; what is important is that the architecture has been able to generate a set of internal states from its interaction with the environment. Furthermore, these states are easily accessible from the outside by looking at the IHU outputs, and could be used for further deliberative purposes

Work. Finish. Publish.
Michael Faraday

9 Conclusions

This section summarizes the main results of this research and how they contribute to answer the questions that have driven this thesis:

1. How can neural network-based controllers be designed for complex robots?
2. Are modular neural networks more suitable than monolithic ones networks for controlling them?
3. How can information about the behaviour to generate be better used? Can we constrain its use?

Obtained results are compared to related works. Additional analysis of the results obtained and a discussion about how they can shed some light on different fields of artificial intelligence are also provided. The chapter ends with a list of future related works.

9.1 ANN's for the control of complex robots

The work elaborated for this thesis provides an answer to the question of how to create neural controllers for complex robots. The approach introduced is based on neuro-evolutionary methods for neural network training. This approach, described in chapter 4, lies in a high level of modularization for the controller going beyond the more usual behaviour-based modularization, and the ability to train these modules separately in a closely related task. The proposed methodology has been validated by testing it in a variety of different robots and different behaviours, both real and simulated, with satisfactory results.

When compared with other architectures (modular and monolithic) in experiments, the DAIR architecture proves to be one of the best. Furthermore, results in chapter 5 show that all of the modular architectures analyzed perform better than their monolithic counterparts. On top of this, as the level of modularization increases, so does the performance. So, the results obtained, even if not providing an entirely conclusive answer to the question of whether to use modular or monolithic systems, do shift the answer in favour of modular

networks being more adequate for the control of complex robots with complex behaviours.

In the DAIR approach,the evolutionary process is completely guided by the information about the required behaviour. All the available information can be used to evolve one single module or combination of them. Due to the characteristics of the architecture, information can be introduced into only those parts of the controller that the designer believes should be affected by it. It has been claimed that the use of information bias in the evolutionary process cannot be avoided due to the great constraints introduced in the evolution of the controller, as stated in chapter 6. Since it looks like the information is required, we make use of it as much as possible and design the process to benifit as much as possible from it.

9.1.1 Advantages of the DAIR method

Our approach is presented as a neural network-based modular architecture for controllers, trained by a neuro-evolutionary learning method, for robotic systems composed of multiple sensors and actuators. Our method has the following advantages:

1. By evolving modules separately, the search space dimension that the evolutionary algorithm has to afford is significatively reduced. This is what staged evolution attempts to accomplish.

2. At each new stage, the newly added modules will start evolving not from a random position in the search space, but from a place provided by the previous stage which should be related to the new task to be evolved, making it easier to obtain the desired behaviour. This is what incremental evolution sets out to accomplish.

The DAIR method therefore combines both techniques (staged evolution and incremental evolution) into a single method, obtaining what we call as *progressive design*. The method is also general in terms of both robot and task.

As was described in chapter 2, staged evolution and incremental evolution were successfully applied separately in other works. Some examples of incremental evolution are [Gomez and Miikkulainen, 1996, Gómez and Miikkulainen, 1999, Yong and Miikkulainen, 2001, Islam et al., 2001] and those for staged evolution [Lara et al., 2001]. Also, some works have designed particular cases of the DAIR approach where both the processes of staged and incremental evolution were performed [Ijspeert, 2001, Hallam and Ijspeert, 2003].

Additionally, the robot controller's current state can be checked at any point in time using the DAIR architecture, simply by looking at its state vector, as described in chapter 7. In some research frameworks, there exists some opposition to the use of ANN's in any environment, mainly because they act as a black box. That is, what the network is performing cannot be inferred from the controller's output (i.e., the action performed by the controller). The DAIR

architecture, even if it does not provide a complete insight, offers us a glimpse into the controller's inner behaviour by allowing access to the state vector.

The main difference with other neural controllers lies in the fact that sensors and actuators outputs are accessible, instead of only having access to the actuation, as provided by other controllers. This *actuation only* feature does not allow one to infer any information of the inner working of the controller. The DAIR architecture, by providing access to its inner state, allows for a better analysis of what is happenning inside the controller. More importantly though, this inner state represents the current situation as the robot is experiencing it. The robot has created its experiences and shows them to us by way of the state vector.

9.1.2 Drawbacks for highly complex robots

DAIR architecture has proven to function well with a complex Aibo robot, where more than 30 modules were required. However, it is difficult to assertain from this experimental result whether the architecture would scale well to robots with hundreds of modules[1]. Two main problems may arise:

1. When the number of modules is so large, it can be difficult to identify precisely which modules to use to start the evolutionary process, and which type of task to assign to them. This problem can be of the same nature as that of identifying which behaviours are required to control a robot when using a behaviour-based architecture. A good knowledge of the task that the robot must accomplish will help in determining the progressive design process.

2. When the number of elements in the controller increases, the newly added modules in the last stage will have to interface with many previously evolved modules. In such cases, two possible situations are devised:

   (a) firstly, what has already been evolved has a stable solution in its m-dimensional search space. Once new modules are added to be evolved and the dimension increases, the number of connections to evolve at that stage may be high, but the controller will start evolution from a previous stable solution. These starting points will make it easier to find solutions if the new task to evolve is close enough. In fact, this effect was observed in section 6.3, when the Khepera robot was evolved in stages. After every successful stage, the next evolutionary stage with an increased number of modules started its fitness with a very high value (see figure 6.9). The same behaviour was observed in the evolution of the Aibo robot when additional controllers for legs were added (see figures 6.27, 6.31 and 6.35).

[1] Even if such robots do not presently exist, they certainly will in the foreseeable future, when robots will come equipped with sensors around the entire body, as well as internal state sensors.

(b) Secondly, due to the high degree of modularity, we are not restricted to a fixed way of evolving the stages. Instead, we can decide which stages best suit us and evolve different parts separately, in order to (once finished their tasks - which may be very different), evolve the connection between them to obtain a final global controller. This procedure was followed in [Lara et al., 2001] and in our own work described in chapter 6 and 7. Further work has to be done to identify whether this connection requires the use of additional neurons to create the interface [Lara et al., 2001]. In our work, additional neurons were not necessary, and it seems that just the additional input connections handled the interface between separately evolved modules.

3. The use of a staged mechanism can lead to sub-optimal solutions. Since the group of IHU's evolved in stage n is frozen from evolving in stage $n+1$, it could result in the solution found in stage $n+1$ not being optimal from the whole controller's point of view. This is the case for decomposable systems, where optimization of one part depends on the actual implementation of the others. However, we will not consider this a limitation, but rather a characteristic of working with systems with limited resources, as happens in nature. This is related to the concept of limited rationality [Simon, 1969]. Furthermore, since IHU's from stage n must add the input connections from the modules added in stage $n+1$, a little tuning and modification of the solution found in stage n is allowed.

9.2 Discussion

This section discusses the use of the DAIR approach as paradigm for the solution of other related problems.

9.2.1 DAIR and scale-up in evolutionary robotics

One of the biggest problems that Evolutionary Robotics faces at present is that of *scaling up*, that is, the use of ER in complex robots. ER has mainly been applied to simple wheeled robots. When the same procedure is applied to more complex robots no behaviour is obtained, as either there is no evolutionary path available, or the search space is so large that it is almost impossible to find a path.

To avoid this pitfall, the DAIR architecture can be introduced as a solution. However, our approach makes a large use of bias, which is not generally desired in evolutionary methods as it drives the evolutionary search towards a specific place within the solutions space; the place where the engineer thinks there may be a solution. According to our study, it is difficult to determine whether the evolutionary method can avoid the introduction of bias if the system to evolve is too restrictive or complex. The introduction of bias is a possible solution to

overcome the lack of an evolutionary path. It would be a directed path though, but a path anyway, where at present there is none.

9.2.2 Tactical modularity for resolution of general problems

Up until this point, the concepts of strategic and tactical modularity have only been applied to the generation of control for robots. However, a step backwards can be taken to gain a wider perspective, and apply those concepts to more general problems where no devices exist, only abstract concepts or variables. Above all, it implies the use of DAIR for the optimization of functions, that is, to define the sub-goals required to generate a goal (strategic modules), and then to create tactical modules for the *elements* that appear in every sub-goal. We understand elements as the *inputs* required to generate the sub-goal, such as the sensors modules, and the *outputs* that define the sub-goal solution, like the actuator modules [Téllez and Angulo, 2008].

The strategic modularity concept has already been applied to such cases for years, even though it was not previously named as such, as it has been shown in the examples described in section 3.2. In these examples the use of tactical modularity can also be applied; in that case, tactical modules can be seen as variables to be optimized.

Example. Let's define the following function,

$$f(x, y, z) = ax + by + cz$$

Four tactical modules can be defined, - one for each of the function variables that obtains the desired values for a, b, c, and another tactical module for the computation of the answer that implements the required $f(x, y, z)$ function. In the same manner as for the control of robots, a knowledge of the domain of application is required, which allows the designer to decide which variables are required to be *controlled* by a module, and what the output of the Modular Neural Networks should be, that is, how the $f(x, y, z)$ function is decomposed.

Applications of these ideas may include typical domains for neural networks like pattern recognition or speech recognition, where tactical modules can be defined for each feature that the application has to detect (sensor), and for each answer that it has to generate (actuator).

9.2.3 Decomposable modularity

When modularity is of the decomposable type, the optimization of one module depends on the optimization of the others (see the definition in the 3.1.1 section). That means that strong couplings exist between the different modules conforming the system. The DAIR approach allows for the evolution of such couplings, and also the optimization of modules depending on the optimization

of previous ones. Even if DAIR modules are independent by themselves, they share dependencies with the rest of modules by means of their connection.

When evolving a group of tactical modules at different stages, modules are independently evolved, but, when added to an existing module or group of them, the connections between them can be seen as an influence of the new module on the existing one, and viceversa. This effect is only intuitively described, and future work will be to mathematically show how these decomposable modules can be optimized in this way. Again, the drawback is that it is not an automatic process, though the designer has to decide which modules to first optimize.

9.2.4 Tactical modularity and the robot inner world

The proposed IHU-based tactical modular architecture can be seen as a dynamical system approach to cognitive robotics using a controlled engineering perspective. Our claim is that the proposed network structure of IHU's provides the autonomous agent with an *inner world* based on internal representations of perception rather than an explicit representational model, following the ideas of internal robotics in [Parisi, 2004] and the double closure scheme in [von Foerster, 1970]. In this architecture the concept of double closure is completely obtained, and sensors and actuators are completely coupled.

9.2.4.1 A control engineering perspective

Feedback control is a simple control structure considering inputs/outputs relationships in a plant (sensorimotor control) [Kuo and Golnaraghi, 2002]. A typical single-input, single-output (SISO) feedback control system is depicted in figure 9.1, for which the *inner world* is defined as the part of the control system corresponding to controller-based units. Similarly, the *outer world* is defined as the part of the control system corresponding to process-based units, i.e., the physical world in which the autonomous agent is situated. From a basic control engineering perspective; in order that the whole system reaches the set point (SP), the control elements (inner world) must be designed using a model that mirrors the outer world as accurately as possible - the so-called *process model*. Controller design procedures in control engineering are traditionally model-based, and so the performance of the whole system depends on how well the process has been modeled: the internal model of the outer world used to generate the inner world must be as exact as possible to the outer world.

A particular element that can help in understanding the concept is the role of the SP. For the effective comparison of the blocks in figure 9.1, the external SP must be translated to an internal SP based on the same units for the controller as for the inner world, e.g., a thermostat translates external SP's from temperature units into voltage units in a range similar to that for the conditioner. It is usually assumed that this conditioning is known to the control engineer designing the control system, so sensors and actuators are considered as part of the process, leading to a clearer control design. However, when this knowledge is not available, sensors and actuators are not predetermined or they are affected

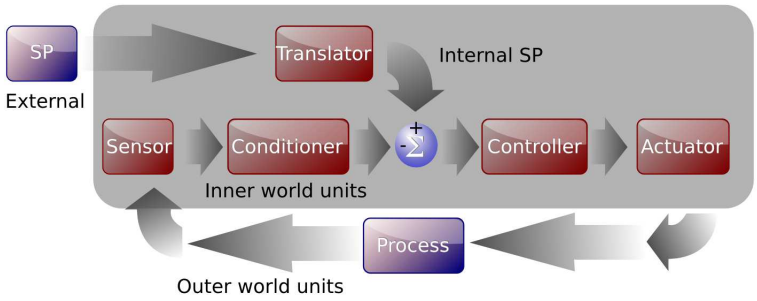


Figure 9.1: A typical SISO feedback control system.

by the environment in an unpredictable manner, and the relationship between the conditioner, controller and translator is no longer a simple additive process. Unlike traditional approaches, a learning procedure or teaching module must exist for designing or modifying the agent's internal representations and intentionality. As observed in figure 9.2, the internal translation of the external SP, which is selected in a certain sense, affects both control elements (conditioner and controller) in an unknown, possibly non-linear manner. The autonomous agent can be defined as the embodiment of *Me*, whereas the part of the autonomous agent that processes information is defined as the *Mind*: control is performed for three elements, the *conditioner*, the *controller* and the *translator*, all of which are directly affected by, or affect the internal SP, i.e., the internal translation of the SP and not the real world SP.

These elements comprising the mind of the autonomous agent are responsible for adapting the relationship between the autonomous agent, that is, the embodiment of the Mind, the environment, and the situation that the agent is currently experiencing:

- The **conditioner** is a control element that adapts what the sensor captures from the outer world to what the mind perceives in its inner world, considering the internal SP.
- The **translator** is a control element that translates the external SP, which is in fact a goal or sub-goal associated with a task, as an interpretation of the outer world. It is a learning function for the whole inner world system.
- The **controller** is a control element that relates internal perception of the outer world, in the form of inner world units, to accomplishment of the task at hand, - interpreted as an internal SP in inner world units. It drives the actuator to change the body-environment situation, and needs continuous, but not exhaustive learning to continually adapt the body to the environment.

Broadly speaking, *Me* depends on the goal (goal-directed training) interpreted by the translator, on the environment (outer world) interpreted by the

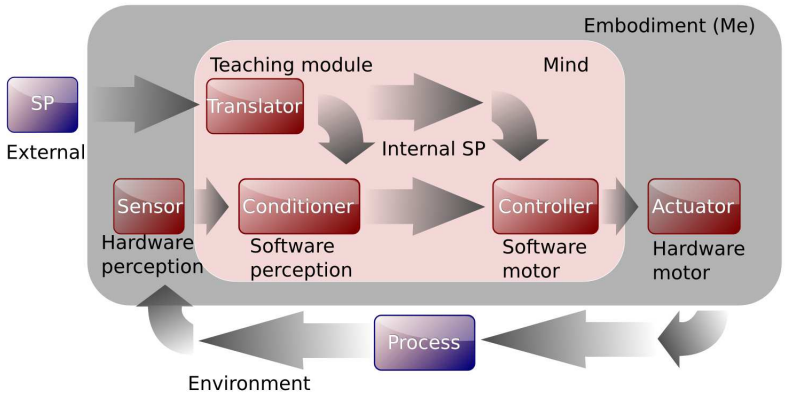


Figure 9.2: *Mind* is embodied in *Me* (the autonomous agent) and is situated in an environment (the process).

conditioner, and on the body (mind, sensor and actuator) acting through the controller. Hence, *Me* is based on the *mind* (translator–conditioner–controller). Information from the environment is *mentally presented*, instead of *mentally represented*: there is no need, as in the traditional approach, to consider any accurate correspondence between the internal model and the real world via a process model. The internal model is built from interaction of the body with the environment; however, in contrast to [Parisi, 2004], it does not try to exactly imitate the world, but is an interpretation of it [Kalveram et al., 2005]. The important point is that the agent's view of the outer world makes sense to the agent itself. Experience and information obtained from the world are therefore highly subjective.

9.2.4.2 Internal model: the *mind–body* problem

From figure 9.2 it is evident that sensor processes (hardware–software perception) and motor processes (software–hardware motion) are separated in "Me". However, feedback from the outer world is not enough to achieve the von Foerster concept of *double closure*:

> "The meanings of the signals of the sensorium are determined by the motorium; and the meanings of the signals of the motorium are determined by the sensorium."

Therefore, perception and motion must be connected to each other in such a form that information has its origin in this creative circle. Motor stimuli must also be sent to the sensor elements to 'predict' what to sense upon real sensation in the outer world.

In terms of control engineering, an internal model control (IMC) structure [Isidori et al., 2003] can be chosen to introduce the concept that an information

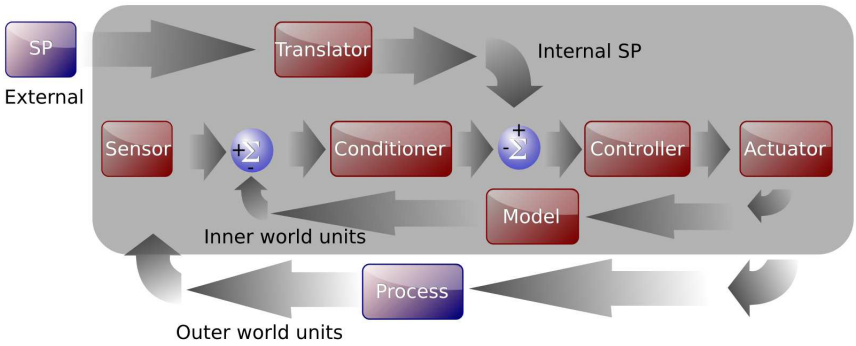


Figure 9.3: Feedback control loop with internal model control.

flow exists from the actuator control signals to the conditioner (see figure 9.3).

These signals model the environment, and hence a *modeler* is defined for modeling the environment and conditioning the outer world to the inner world units. The inner signals sent by the controller are fed back to the modeler instead of real world signals from the actuator, since this structure does not aim to exactly model the world, but to obtain a subjective internal representation of the outer world. Our answer to the *mind-body* separation problem is shown in figure 9.4 using the IHU control concept, i.e., mind is based on the sensors and actuators, which are governed by a learning module that translates external SP's into the driving of fitness functions.

Extension of the proposed SISO control to a typical multi-input, multi-output (MIMO) system results in a control system exactly the same as our proposed network of IHU's (figure 9.4).

9.2.4.3 The translator

As argued by [Cañamero, 2005], current embodied approaches are not well enough developed to be able to model *higher-level* aspects of behaviours. In this sense, our tactical modular concept represents a new reactive interpretation of the mind based on internal representations of the real world for its design, i.e. the control elements, to successfully carry out a task. The translator module, converted into a teaching module external to the decentralized reactive mind, contains the fitness functions associated with the tasks that drive learning in the modeler and controller modules. These latter modules that translated from/to the world signals to/from the internal representations are usually neural networks. Ideally, deliberative control, which is outside the scope of this study, must be involved in the translator. Hence, fitness functions related to the task at hand have been assumed to be known to the expert.

Although our proposed architecture is focused on the emergence of behaviour and not on deliberative interpretation of the mind, it can facilitate the integration of both reactive and deliberative controls in two forms. Firstly, use of our

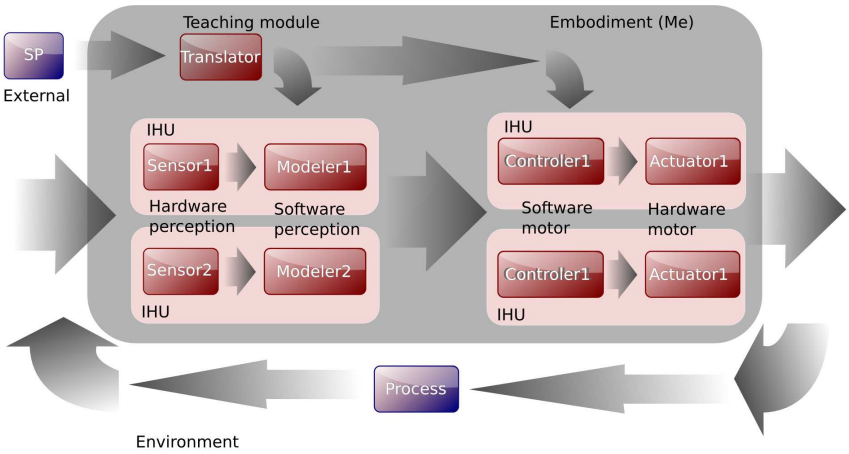


Figure 9.4: The "Mind" designed through collaborative IHU's in the form of a MIMO and the decentralized control architecture.

tactical modularity is not in opposition to the use of the strategic approach, in fact, use of both types of modularity in the same controller could well be the required solution for complex behaviours in complex robots. Thus, for example, it is possible to merge our physical device-based neural network tactical modules with the strategic modules used in [Nolfi, 1997], who demonstrated that module switching and interaction are correlated with low-level sensorimotor mappings.

Secondly, autonomous agents (robots) beyond those completely reactive to the environment can be obtained using recurrent neural networks, i.e., with internal-feedback connections (internal states) in the modeler and controller modules of each IHU, so that they can initiate action independent of the immediate situation, and organize behaviour in anticipation of future events.

9.3 Future work

This section contains a list of new lines of research that will be open to continue build on the work developed in the thesis. Apart from small parts which have been indicated along the thesis text, there are three main subjects that can be advised as possible lines of research. All the points discussed arose during the creation of the thesis, and were left aside because they were too far from the main line of research, or because there wasn't sufficient time to explore them all properly.

9.3.1 Tactical modularity as a walking reflex system

Tactical modularity was applied in chapter 6 to the generation of a walking Aibo robot. The walking application was only selected as an application example of

the architecture, but it showed how sensors and actuators could be coupled for the generation of behaviours. Some authors have suggested that this strong coupling between sensors and actuators in a walking system would produce a reflex system for walking machines [ref. Jun Nishii, personal communication, Jan, 2006]. To date, most of the walking mechanisms for robots implement an almost open loop walking mechanism, that is, sensors are very lightly used to correct the walking pattern on the fly [Kajita et al., 2003]. Hence, the robot is not able to react adequately when faced with unpredicted situations in the walking system, and is only able to walk on a limited number of surfaces in certain environments, - only those which meet the requirements of the walker. To improve walking behaviours, researchers have attempted to integrate a reflex system into the walking machine that should permit the robot to react to small perturbances to its walking [Ijspeert, 2002, Righetti and Ijspeert, 2008]. This reflex system is usually an external separated component that takes the sensor lectures as inputs and provides the walker with some control signal to modify its behaviour accordingly.

Tactical modularity can be used to implement a completely coupled walking system between sensors and actuators, where the reflex system would be embedded into the walking mechanism. The walking system, mainly driven by the actuator signals, would have a directly coupled reflex system which is not a separated part of the walking, but rather an integrated part of it. It is suggested that animals have such type of a walking mechanism in order to improve their walking behaviour [Pearson and Gordon, 2000].

9.3.2 Deliberative control

IHU's have been shown to be able to modify their behaviour depending on the value of an external tonic signal (see chapter 7). On top of that, chapter 8 is devoted to show how the tactical modules are able to *express* their status, and that of the robot they are controlling through the state vector. It is then possible to design a higher control layer which reads the current state of the reactive tactical structure, and then decides how to modify its behaviour through the use of a tonic signal. Deliberation of the higher layer would be based on the performance and current task required for the robot. This would include the capacity to deliberately control complex robots using only ANN's in the DAIR architecture .

The deliberative layer can be created by any already existing artificial intelligence method: an agent, an error minimizing function, a look-up table, etc... However, a most interesting approach would be if the deliberation is performed as an additional layer of tactical modules, allowing the scaling up of the same concept architecture (see figure 9.5). In such a situation, a first reactive layer of tactical modules would take the basic reactive control of the robot. The modules in such a layer; let's call them TR-IHU's, for Tactical-Reactive IHU's, would learn a basic behaviour for the robot and how to slightly modify it according to the value of a tonic signal, which may be either the same, or different for every module. Once the reactive layer is trained in such a way, the delibera-

tive would have to be trained. An additional layer of tactical modules can be added for the control of the TR-IHU layer. This additional layer should learn to modify the behaviour of the TR-IHU layer accordingly with the current task at hand. This layer should deliberate what it's current status is, and formulate a plan to achieve its goal. We will call this layer the TD-IHU's layer, for Tactical-Deliberative.

An example is the T-Maze experiment in [Bergfeldt and Linåker, 2002] or in [Blynel and Floreano, 2003], which has been adapted by us for the Aibo robot (see figure 9.6). In the T-maze, the robot walks along a straight corridor; at some moment, the robot will see a red or blue dot on the wall in front of it; the dot will only be shown for a second. The robot has to learn that once it reaches the crossroad, it has to turn to one side: left if the dot is red or right if the dot is blue. Once the TR-IHU layer is trained in walking, the TD-IHU layer has to learn to memorize and decide, and to use the TR-IHU layer for its purpose.

In such an experiment, using the DAIR structure, the TR-IHU layer should learn to walk, and, due to the different values of the tonic signal, vary the speed of movement of its legs (as in chapter 7) so that it should allow the stop, start and turn behaviours from a single TR-IHU layer with different TOC values. The TD-IHU layer should then learn to control the TR-IHU layer accordingly to solve the T-Maze problem, by reading the state vector and setting the required TOC values accordingly .

Deliberative control can include either the control to slightly modify the behaviour of a given reactive layer (that is, to modify the behaviour of the tactical modules that compose a strategic module), or the control to select which strategic module should be activated (in a system composed of strategic and tactical modules).

9.3.3 Liar IHU's

To date, IHU modules have been desgined with a single output, which is used either for action in actuator-IHU's, or as a processed sensor value in sensor-IHU's. However, it could be interesting to investigate whether the architecture improves in both learning rate and fitness value if each IHU is allowed to have two or more outputs. The IHU would use one output for its related purpose, and the second one to communicate a, perhaps, different bit of information to the rest of IHU's.

It would be interesting to observe whether such types of connections will generate different outputs in the same IHU, giving place to a kind of *liar* IHU where it says one thing but does another. Intuitively, two different outputs may increase the flexibility of the architecture, since it is not clear that the actuator output that is required to perform a given action by the robot is the best information to communicate to the other IHU's.

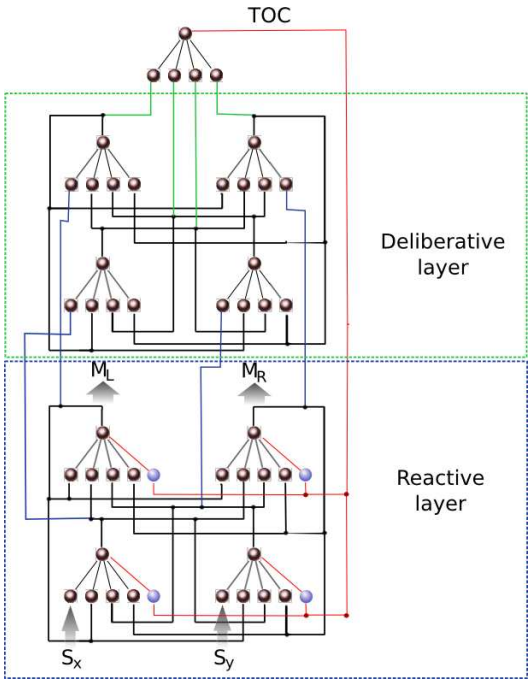


Figure 9.5: A deliberative structure using tactical modules.

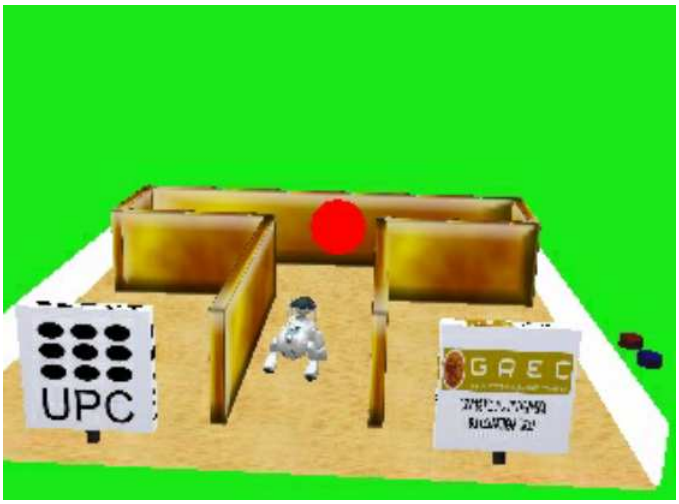


Figure 9.6: T-Maze simulation created to implement a TD-IHU's solution for the Aibo robot (by Francesc Espasa).

Bibliography

[Albus et al., 1987] Albus, J., McCain, H., and Lumia, R. (1987). Nasa/nbs standard reference model for telerobot control system architecture (nasrem). Technical note 1235, Robot system divisions, National bureau of standards.

[Alpaydin, 1993] Alpaydin, E. (1993). Multiple networks for function learning. In *Proceedings of Int. conf. on Neural Networks*.

[Alpaydin and Jordan, 1996] Alpaydin, E. and Jordan, M. (1996). Local linear perceptrons for classification. *IEEE Transactions on neural networks*, 7(3):788–792.

[Anderson, 1989] Anderson, C. (1989). Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9(3):31–37.

[Angulo and Téllez, 2004] Angulo, C. and Téllez, R. (2004). Distributed intelligence for smart home appliances. In Raúl Giráldez, José C. Riquelme, J. S. A.-R., editor, *Tendencias de la Minería de Datos en España.*, pages 1–12.

[Arbib, 1992] Arbib, M. (1992). *The Encyclopedia of Artificial Intelligence (2 nd Ed)*, chapter Schema Theory, pages 1427–1443. Wiley Interscience.

[Arbib, 1995] Arbib, M. (1995). *Handbook of brain theory and neural networks*. The MIT Press.

[Arkin, 1998] Arkin, R. (1998). *Behavior-based robotics*. MIT Press.

[Auda, 1996] Auda, G. (1996). *Cooperative modular neural network classifiers*. PhD thesis, University of Waterloo.

[Auda and Kamel, 1997a] Auda, G. and Kamel, M. (1997a). Cmnn: Cooperative modular neural networks for pattern recognition. In *Pattern Recognition in Practice V Conference*.

[Auda and Kamel, 1997b] Auda, G. and Kamel, M. (1997b). Modular neural network classifiers: a comparative study. In *Proceedings of the Int. Conf. Neural Networks Appli.*

[Auda and Kamel, 1999] Auda, G. and Kamel, M. (1999). Modular neural networks: a survey. *International Journal of Neural Systems*, 9(2):129–151.

[Azam, 2000] Azam, F. (2000). *Biologically inspired modular neural networks*. PhD thesis, Virginia Polytechnic Institute and State University.

[Ballard, 1986] Ballard, D. (1986). Cortical connections and parallel processing. *The behavioral and Brain sciences*, 9:279–284.

[Barto et al., 1983] Barto, A., Sutton, R., and Anderson, C. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13(5):834–846.

[Battiti and Colla, 1994] Battiti, R. and Colla, A. (1994). Democracy in neural nets: voting schemes for classification. *Neural networks*, 7(4):69–707.

[Beer, 1995] Beer, R. (1995). On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3(4):469–509.

[Beer and Gallagher, 1992] Beer, R. and Gallagher, J. (1992). Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1(1):91–122.

[Bekey, 2005] Bekey, G. (2005). *Autonomous robots. From biological inspiration to implementation and control*. The MIT Press.

[Bergfeldt and Linåker, 2002] Bergfeldt, N. and Linåker, F. (2002). Self-organized modulation of a neural robot controller. In *Proceedings of the International Joint Conference on Neural Networks*.

[Bianco and Nolfi, 2004] Bianco, R. and Nolfi, S. (2004). Evolving the neural controller for a robotic arm able to grasp objects on the basis of tactile sensors. *Adaptive Behavior*, 12(1):37–45.

[Billard and Ijspeert, 2000] Billard, A. and Ijspeert, A. J. (2000). Biologically inspired neural controllers for a motor control in a quadruped robot. In *Proceedings of the International Joint Conference on Neural Network*.

[Bishop, 1995] Bishop, C. M. (1995). *Neural networks for pattern recognition*. Oxford University Press.

[Blynel and Floreano, 2003] Blynel, J. and Floreano, D. (2003). Exploring the t-maze: Evolving learning-like robot behaviors using ctrnns. In *Proceedings of the EvoRobot*.

[Boers and Kuiper, 1992] Boers, E. J. W. and Kuiper, H. (1992). Biological metaphors and the design of modular artificial neural networks. Master's thesis, Leiden University.

[Bongard, 2002] Bongard, J. (2002). Evolving modular genetic regulatory networks. In *Proceedings of CEC*.

[Bongard, 2003] Bongard, J. (2003). *Incremental approaches to the combined evolution of a robot's body and mind*. PhD thesis, University of Zurich.

[Bongard and Pfeifer, 2003] Bongard, J. C. and Pfeifer, R. (2003). Evolving complete agents using artificial ontogeny. In Hara, F. and Pfeifer, R., editors, *Morpho-functional Machines: The New Species (Designing Embodied Intelligence)*, pages 237–258. Springer-Verlag.

[Braitenberg, 1984] Braitenberg, V. (1984). *Vehicles*. MA: MIT Press.

[Brooks, 1986] Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23.

[Brooks, 1991] Brooks, R. (1991). Intelligence without representation. *Artificial Intelligence*, 47:139–159.

[Brooks and Stein, 1994] Brooks, R. A. and Stein, L. A. (1994). Building brains for bodies. *Autonomous Robots*, 1:7–25.

[Buessler and Urban, 2003] Buessler, J. and Urban, J. (2003). *Biologically inspired robot behavior engineering*, chapter Modular neural architectures for robotics, pages 261–298. Physica-Verlag.

[Buessler et al., 2002] Buessler, J.-L., Urban, J.-P., and Gresser, J. (2002). Additive composition of supervised self-organized maps. *Neural Processing Letters*, 15:9–20.

[Caelli et al., 1999] Caelli, T., Guan, L., and Wen, W. (1999). Modularity in neural computing. In *Proceedings of the IEEE*, volume 87, pages 1497–1518.

[Calabretta et al., 2003] Calabretta, R., Di Ferdinando, A., Wagner, G. P., and Parisi, D. (2003). What does it take to evolve behaviorally complex organisms? *BioSystems*, 69:245–262.

[Calabretta et al., 1998] Calabretta, R., Nolfi, S., Parisi, D., and Wagner, G. (1998). A case study of the evolution of modularity: Towards a bridge between evolutionary biology, artificial life, neuro and cognitive science. In C. Adami, R. Belew, H. K. and Taylor, C., editors, *Proceedings of Artificial Life VI*. MA: MIT Press.

[Calabretta et al., 2000] Calabretta, R., Nolfi, S., Parisi, D., and Wagner, G. (2000). Duplication of modules facilitates functional specialization. *Artificial Life*, 1(6):69–84.

[Calabretta and Parisi, 2005] Calabretta, R. and Parisi, D. (2005). Evolutionary connectionism and mind/brain modularity. In Rasskin-Gutman, W. C. . D., editor, *Modularity. Understanding the development and evolution of complex natural systems*, pages 309–330. The MIT Press.

[Calancie et al., 1994] Calancie, B., Needham-Shropshire, B., Jacobs, P., Willer, K., Zych, G., and Green, B. A. (1994). Involuntary stepping after chronic spinal cord injury. evidence for a central rhythm generator for locomotion in man. *Brain*, 117(5):1143–1159.

[Callebaut, 2005] Callebaut, W. (2005). *Modularity. Understanding the Development and Evolution of Natural Complex Systems*, chapter The ubiquity of modularity. The MIT Press.

[Carruthers, 2004] Carruthers, P. (2004). *Contemporary Debates in the Philosophy of Science*, chapter The mind is a system of modules shaped by natural selection. Blackwell.

[Carruthers, 2005] Carruthers, P. (2005). *Contemporary Debates in Cognitive Science*, chapter The case for massively modular models of mind. Blackwell.

[Cañamero, 2005] Cañamero, L. (2005). Emotion understanding from the perspective of autonomous robots research. *Neural Networks*, 18(4):445–455.

[Chen and Chi, 1999] Chen, K. and Chi, H. (1999). A modular neural network architecture for pattern classification based on different feature sets. *International Journal of Neural Systems*, 9(6):563–581.

[Cho and Shimohara, 1997] Cho, S. B. and Shimohara, K. (1997). Emergence of structure and function in evolutionary modular neural networks. In *Proceedings of the Fourth European Conference on Artificial Life*, pages 197–204.

[Choe and Bhamidipati, 2004] Choe, Y. and Bhamidipati, S. (2004). Autonomous acquisition of the meaning of sensory states through sensory-invariance driven action. In Ijspeert, A. J., Murata, M., and Wakamiya, N., editors, *Biologically Inspired Approaches to Advanced Information Technology, Lecture Notes in Computer Science*, volume 3141, pages 176–188. Springer.

[Collins and Richmond, 1994] Collins, J. and Richmond, S. (1994). Hard-wired central pattern generators for quadrupedal locomotion. *Biological Cybernetics*, 71:375–385.

[Collins and Stewart, 1993] Collins, J. J. and Stewart, I. N. (1993). Coupled nonlinear oscillators and the symmetries of animal gaits. *Journal of Nonlinear Science*, 3(1):349–392.

[Colombetti and Dorigo, 1992] Colombetti, M. and Dorigo, M. (1992). Learning to control an auonomous robot by distributed genetic algorithms. In *From Animals to Animats 2, Proceedings of the 2nd International Conference on the Simulation of Adaptive Behavior*.

[Corwin et al., 1994] Corwin, E., Greni, S., Logar, A., and Whitehead, K. (1994). A multi-stage neural network classifier. In *Proceedings of World congress on neural networks*.

[Cyberbotics, 2005] Cyberbotics (2005). Webots refernce manual, release 5.0.1.

[Davis, 1996] Davis, I. (1996). *A Modular Neural Network Approach to Autonomous Navigation*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

[Dawkins, 1987] Dawkins, R. (1987). *The blind watchmaker*. Norton.

[Dawkins and Krebs, 1979] Dawkins, R. and Krebs, J. (1979). Arms races between and within species. In *Proceedings of the Royal Society of London B*, volume 205, pages 489–511.

[de Bollivier et al., 1991] de Bollivier, M., Gallinari, P., and Thiria, S. (1991). Cooperation of neural nets for robust classification. In *Proc. Internat. Joint Conference on Neural Networks*.

[De Jong et al., 2004] De Jong, E., Thierens, D., and Watson, R. (2004). Defining modularity, hierarchy, and repetition. In *Proceedings of the GECCO Workshop on Modularity, regularity and hierarchy in open-ended evolutionary computation*, pages 2–6.

[De Sa and Balard, 1998] De Sa, V. and Balard, D. (1998). Category learning through multi-modality sensing. *Neural computation*, 10(5):1097–1117.

[Dewey, 1896] Dewey, J. (1896). The reflex arc concept in psychology. *Psych. Rev.*, 3:357–370.

[Di Ferdinando and Parisi, 2000] Di Ferdinando, A. Calabretta, R. and Parisi, D. (2000). Evolving modular architectures for neural networks. In *Proceedings of the sixth Neural Computation and Psychology Workshop: Evolution, Learning and Development*.

[Doncieux and Meyer, 2004] Doncieux, S. and Meyer, J.-A. (2004). Evolution of neurocontrollers for complex systems: alternatives to the incremental approach. In *Proceedings of The International Conference on Artificial Intelligence and Applications*.

[Dorigo, 1995] Dorigo, M. (1995). Alecsys and the autonomouse: Learning to control a real robot by distributed classifier systems. *Machine learning*, 19(3):209–240.

[Dorigo and Colombetti, 1998] Dorigo, M. and Colombetti, M. (1998). *Robot shaping: an experiment in behavior engineering*. The MIT Press.

[Dorigo et al., 2004] Dorigo, M., Trianni, V., Sahin, E., Gro, R., Labella, T. H., Baldassarre, G., Nolfi, S., Deneubourg, J.-L., Mondada, F., Floreano, D., and Gambardella, L. M. (2004). Evolving self-organizing behaviors for a swarm-bot. *Autonomous Robots*, 17(2–3):223–245.

[Elman, 1991] Elman, J. (1991). Incremental learning, or the importance of starting small. In *Proceedings of the 13th Annual Conference of the Cognitive Science Society*, pages 443–448.

[Filliat et al., 1999] Filliat, D., Kodjabachian, J., and a. Meyer, J. (1999). Incremental evolution of neural controllers for navigation in a 6 legged robot. In *Proc. of the Fourth International Symposium on Artificial Life and Robotics*, pages 745–750. Univ. Press.

[Floreano and Mondada, 1994] Floreano, D. and Mondada, F. (1994). Automatic creation of an autonomous agent: Genetic evolution of a neural network driven robot. In *From Animals to Animats III*. MA: MIT Press.

[Floreano and Mondada, 1996] Floreano, D. and Mondada, F. (1996). Evolution of homing navigation in a real mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26:396–407.

[Floreano et al., 2001] Floreano, D., Nolfi, S., and Mondada, F. (2001). Co-evolution and ontogenic change in competing robots. In Patel, M., Hanover, V., and Balakrishnan, K., editors, *Advances in the evolutionary synthesis of intelligent agents*. MIT Press.

[Fodor, 1983] Fodor, J. (1983). *The modularity of mind*. Cambridge, MA: MIT Press.

[Fodor, 2000] Fodor, J. (2000). *The mind doesn't work that way: the scope and limits of computational psychology*. MIT Press.

[Fujita, 2001] Fujita, M. (2001). Aibo: Toward the era of digital creatures. *The International Journal of Robotics Research*, 20:781–794.

[Fujita and Kitano, 1998] Fujita, M. and Kitano, H. (1998). Development of an autonomous quadruped robot for robot entertainment. *Autonomous robots*, 5(1):7–18.

[Funes and Pollack, 1999] Funes, P. and Pollack, J. (1999). Computer evolution of buildable objects. In *Evolutionary Design by Computers*. Morgan Kaufmann.

[Gallagher et al., 1996] Gallagher, J. C., Beer, R. D., Espenschield, K. S., and Quinn, R. D. (1996). Application of evolved locomotion controllers to a hexapod robot. *Robotics and Autonomous Systems*, 19:95–103.

[Garibay et al., 2004] Garibay, I., Garibay, O., and Wu, A. (2004). Effects of module encapsulation in repetitively modular genotypes on the search space. In *Proceedings of Genetic and Evolutionary Computation Conference - GECCO 2004*, volume 1, pages 1125–1137.

[Gatt, 1993] Gatt, E. (1993). On the role of stored internal state in the control of autonomous mobile robots. *AI Magazine*, Spring:64–73.

[Geshwind and Galaburda, 1987] Geshwind, N. and Galaburda, A. (1987). *Cereberal lateralization: biological mechanisms, associations and pathology*. The MIT Press.

[Ghahramani, 1995] Ghahramani, Z. (1995). *Computation and psychophysics of sensorimotor integration*. PhD thesis, M.I.T.

[Ghahramani, 2004] Ghahramani, Z. (2004). *Advanced Lectures in Machine Learning. Lecture Notes in Computer Science*, volume 3176, chapter Unsupervised Learning, pages 72–112. Springer-Verlag.

[Ghahramani et al., 1997] Ghahramani, Z., Wolpert, D. M., and Jordan, M. I. (1997). Computational models of sensorimotor integration. In Morasso, P. G. and Sanguineti, V., editors, *Self-Organization, Computational Maps and Motor Control*, pages 117–147. Elsevier.

[Gomez and Miikkulainen, 1996] Gomez, F. and Miikkulainen, R. (1996). Incremental evolution of complex general behavior. Technical Report AI96-248, University of Texas.

[Gomez and Schmidhuber, 2005] Gomez, F. and Schmidhuber, J. (2005). Coevolving recurrent neurons learn deep memory pomdps. In *Proc. of the 2005 conference on genetic and evolutionary computation (GECCO)*.

[Graham and Oppacher, 2007] Graham, L. and Oppacher, F. (2007). A multiple-function toy model of exaptation in a genetic algorithm. In *IEEE Congress on Evolutionary Computation*.

[Grillner, 1985] Grillner, S. (1985). Neurobiological bases of rhythmic motor acts in vertebrates. *Science*, 228:143–149.

[Gruau, 1994] Gruau, F. (1994). *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France.

[Gruau, 1995] Gruau, F. (1995). Automatic definition of modular neural networks. *Adaptive Behaviour*, 3(2):151–183.

[Gruau, 1997] Gruau, F. (1997). Cellular encoding for interactive evolutionary robotics. In *Proceedings of the 4th european conference on artificial life*.

[Gruau et al., 1996] Gruau, F., Whitley, D., and L.Pyeatt (1996). A comparison between celular encoding and direct encoding for genetic neural networks. In *Genetic Programming 1996: Proceedings of the First Annual Conference*. MA: MIT Press.

[Gunderson and Gunderson, 2009] Gunderson, L. and Gunderson, J. (2009). *Robots, reasoning and reification.* Springer.

[Gómez and Miikkulainen, 1999] Gómez, F. and Miikkulainen, R. (1999). Solving non-markovian control tasks with neuroevolution. In *Proceedings of the IJCAI99*.

[Hackbarth and Mantel, 1991] Hackbarth, H. and Mantel, J. (1991). Modular connectionist structure for 100-word recognition. In *Proc. Internat. Joint Conf. on Neural networks*.

[Hallam and Ijspeert, 2003] Hallam, J. and Ijspeert, A. (2003). *Using evolutionary methods to parametrize neural models: a study of the lamprey central pattern generator*, chapter 4, pages 119–142. Physica-Verlag.

[Hampshire and Waibel, 1989] Hampshire, J. and Waibel, A. (1989). The meta-pi network:building distributed knowledge representations for robust pattern recognition. 1989. Technical Report CMU-CS-89-166, Carnegie Mellon University.

[Hansen and Salamon, 1990] Hansen, L. and Salamon, P. (1990). Neural network ensembles. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12(10):993–1001.

[Hermann and Urban, 2003] Hermann, G.and Wira, P. and Urban, J.-P. (2003). Neural networks organizations to learn complex robotic functions. In *Proceedings of the European Symposium on Artificial Neural Networks*, pages 33–38.

[Holh et al., 2006] Holh, L., Téllez, R., Michel, O., and Ijspeert, A. (2006). Aibo and webots: Simulation, wireless remote control and controller transfer. *Robotics and autonomous systems*, 54(6):472–485.

[Holland, 1987] Holland, J. (1987). Genetic algorithms and classifier systems: foundations and future directions. In *Proceedings of the 2nd international conference on genetic algorithms*.

[Hopfield, 1984] Hopfield, J. (1984). Neurons with graded response properties have collective computational properties like those of two-state neurons. In *Proc. National Academy of Sciences USA*, volume 81, pages 3088–3092.

[Hornby, 2003] Hornby, G. (2003). *Generative Representations for Evolutionary Design Automation*. PhD thesis, Brandeis University Dept. of Computer Science.

[Hornby et al., 2003] Hornby, G., Lipson, H., and Pollack, J. (2003). Generative representations for the automated design of modular physical robots. *IEEE Transactions on robotics and automation*, 19(4):703–719.

[Hornby and Pollack, 2002] Hornby, G. and Pollack, J. (2002). Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8:223–246.

[Husbands et al., 1995] Husbands, P., Harvey, and I. Cliff, D. (1995). Circle in the round: State space attractors for evolved sighted robots. *Robotics and Autonomous Systems*, 15:83–106.

[Husken et al., 2002] Husken, M., Igel, C., and Toussaint, M. (2002). Task-dependent evolution of modularity in neural networks. *Connection science*, 14(3):219–229.

[Ijspeert, 1998] Ijspeert, A. (1998). *Design of artificial neural oscillatory circuits for the control of lamprey- and salamander-like locomotion using evolutionary algorithms*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh.

[Ijspeert, 2002] Ijspeert, A. (2002). *The Handbook of Brain Theory and Neural Networks, 2nd Edition*, chapter Locomotion, vertebrate, pages 649–654. Bradford Books.

[Ijspeert, 2001] Ijspeert, A. J. (2001). A connectionist central pattern generator for the aquatic and terrestrial gaits of a simulated salamander. *Biological Cybernetics*, 84:331–348.

[Isidori et al., 2003] Isidori, A., Marconi, L., and Serrani, A. (2003). *Robust autonomous guidance: an internal model approach*. Springer.

[Islam et al., 2001] Islam, M., Terao, S., and Murase, K. (2001). Incremental evolution of autonomous robots for a complex task. In *4th International Conference ICES*.

[Jacobs, 1990] Jacobs, R. (1990). *Task decomposition through competition in a modular connectionist architecture*. PhD thesis, University of Massachusets, Amherst, MA, USA.

[Jacobs, 1995] Jacobs, R. (1995). Methods of combining experts' probability assessments. *Neural computation*, 7:867–888.

[Jacobs and Jordan, 1993] Jacobs, R. and Jordan, M. (1993). Learning piecewise control strategies in a modular neural network architecture. *IEEE trans. syst. man cybern.*, 23(2):337–345.

[Jacobs et al., 1991a] Jacobs, R., Jordan, M., and Barto, A. (1991a). Task decomposition through competition in a modular connectionist architecture: the what and where vision tasks. *Cognitive Science*, 15:219–250.

[Jacobs et al., 1991b] Jacobs, R., Jordan, M., Nowlan, S., and Hinton, G. E. (1991b). Adaptive mixture of local experts. *Neural Computation*, 1(3):79–87.

[Jakobi, 1998] Jakobi, N. (1998). Minimal simulations for evolutionary robotics.

[Jordan and Jacobs, 1994] Jordan, M. and Jacobs, R. (1994). Hierarchical mixtures of experts and em algorithm. *Neural computation*, 6:181–214.

[Kajita et al., 2003] Kajita, S., Kanehiro, F., Kaneko, K., Fujiwara, K., Harada, K., Yokoi, K., and Hirukawa, H. (2003). Biped walking pattern generation by using preview control of zero-moment point. In *Proceedings of the International Conference on Robotics & Automation*.

[Kalveram et al., 2005] Kalveram, K., Schinauer, T., Beirle, S., and Jansen-Osmann, P. (2005). Threading neural feedforward into a mechanical spring: how biology exploits physics in limb control. *Biological Cybernetics*, 92(4):229–240.

[Kaplan, 2005] Kaplan, F. (2005). Simple models of distributed co-ordination. *Connection Science*, 17(3-4):249–270.

[Kimura et al., 1999] Kimura, H., Akiyama, S., and Sakurama, K. (1999). Realization of dinamic walking and running of the quadruped using neural oscillator. *Autonomous Robots*, 7(3):247–258.

[Kimura et al., 2001] Kimura, H., Fukuoka, Y., and Konaga, K. (2001). Adaptive dynamic walking of a quadruped robot by using neural system model. *Advanced Robot*, 15:859–876.

[Klavins et al., 2001] Klavins, E., Komsuoglu, H., Full, R. J., and Koditschek, D. E. (2001). *Dynamical Legged Locomotion, in Neurotechnology for Biomimetic Robots*, chapter The role of reflexes versus central pattern generators in dynamical legged locomotion. In Neurotechnology for Biomimetic Robots. The MIT press.

[Kotsiantis, 2007] Kotsiantis, S. (2007). Supervised machine learning: A review of classification techniques. *Informatica Journal*, 31:249–268.

[Koza, 1991] Koza, J. (1991). Evolution and co-evolution of computer programs to control independently acting agents. In *From animals to animats: Proceedings of the 1st international conference on simulation of adaptive behavior*.

[Krogh and Vedelsby, 1995] Krogh, A. and Vedelsby, J. (1995). Neural network ensembles,cross validation, and active learning. *Advances in Neural Information Processing Systems*, 7:231–238.

[Kuo and Golnaraghi, 2002] Kuo, B. and Golnaraghi, F. (2002). *Automatic control systems*. John Wiley & sons.

[Lara et al., 2001] Lara, B., Hülse, M., and Pasemann, F. (2001). Evolving neuro-modules and their interfaces to control autonomous robots. In *Proceedings of the 5th World Multi-conference on Systems, Cyberbetics and Informatics*.

[Lewis, 2002] Lewis, M. (2002). Gait adaptation in a quadruped robot. *Autonomous robots*, 12(3):301–312.

[Lewis et al., 1992] Lewis, M., Fagg, A., and Solidum, A. (1992). Genetic programming approach to the construction of a neural network control of a walking robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

[Linaker and Niklasson, 2000] Linaker, F. and Niklasson, L. (2000). Time series segmentation using an adaptive resource allocating vector quantization network based on change detection. In *Proceedings of the International Joint Conference on Neural Networks*.

[Lipson and Pollack, 2000] Lipson, H. and Pollack, J. (2000). Automatic design and manufacture of robotic lifeforms. *Nature*, 406(6799):974–978.

[Liu and Yao, 1997] Liu, Y. and Yao, X. (1997). Evolving modular neural networks which generalise well. In *Proceedings of the 1997 International Conference on Evolutionary Computation*.

[Lund and Hallam, 1996] Lund, H. and Hallam, J. (1996). Sufficient neurocontrollers can be surprisingly simple. Technical Report 824, Department of artificial intelligence, University of Edinburgh.

[Manoonpong et al., 2005] Manoonpong, P., Pasemann, F., and Fischer, J. (2005). Modular neural control for a reactive behavior of walking machines. In *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation*.

[Manoonpong et al., 2007] Manoonpong, P., Pasemann, F., and Roth, H. (2007). Modular reactive neurocontrol for biologically-inspired walking machines. *International Journal of Robotics Research*, 26(3):301–331.

[Markelic, 2005] Markelic, I. (2005). Evolving competitive walking behaviour for a quadruped walking machine. Master's thesis, Fraunhofer AiS.

[Mataric and Cliff, 1996] Mataric, M. and Cliff, D. (1996). Challenges for evolving controllers for physical robots. *Robotics and autonomous systems*, 19(1):67–83.

[Mataric, 1992] Mataric, M. J. (1992). Behavior-based control: Main properties and implications. In *Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*, pages 46–54.

[McCain, 2003] McCain, R. (2003). *Game Theory : A Non-Technical Introduction to the Analysis of Strategy*. South-Western College Pub.

[Meuleau et al., 1999] Meuleau, N., Peshkin, L., Kim, K.-E., and Kaelbling, L. (1999). Learning finite state controllers for partially observable environments. In *Proceedings of the 15th International Conference of Uncertainty in Artificial Intelligence*.

[Michel, 2004] Michel, O. (2004). Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42.

[Minsky, 1988] Minsky, M. (1988). *The Society of Mind*. Touchtone Books.

[Mitchell, 1996] Mitchell, M. (1996). *An introduction to genetic algorithms*. MA: MIT Press.

[Mojon, 2004] Mojon, S. (2004). Using nonlinear oscillators to control the locomotion of a simulated biped robot. Master's thesis, École Polytechnique Fédérale de Lausanne.

[Montebelli et al., 2008] Montebelli, A., Herrera, C., and Ziemke, T. (2008). On cognition as dynamical coupling: An analysis of behavioral attractor dynamics. *Adaptive Behavior*, 16(2-3):182–195.

[Moriarty, 1997] Moriarty, D. (1997). *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. PhD thesis, Department of computer sciences, The University of Texas at Austin.

[Moriarty and Miikkulainen, 1996a] Moriarty, D. and Miikkulainen, R. (1996a). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(1-3):11–32.

[Moriarty and Miikkulainen, 1996b] Moriarty, D. and Miikkulainen, R. (1996b). Evolving obstacle avoidance behavior in a robot arm. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*.

[Moriarty and Miikkulainen, 1998] Moriarty, D. and Miikkulainen, R. (1998). Forming neural networks trough efficient and adaptative co-evolution. *Evolutionary Computation*, 5:373–399.

[Mouret and Doncieux, 2008] Mouret, J. and Doncieux, S. (2008). Incremental evolution of animats' behaviors as a multi-objective optimization. In *From Animals to Animats 10*.

[Mouret and Doncieux, 2009a] Mouret, J.-B. and Doncieux, S. (2009a). Evolving modular neural-networks through exaptation. In *IEEE Congress on Evolutionary Computation*.

[Mouret and Doncieux, 2009b] Mouret, J.-B. and Doncieux, S. (2009b). Overcoming the bootstrap problem in evolutionary robotics using behavioral diversity. In *IEEE Congress on Evolutionary Computation*.

[Murino, 1998] Murino, V. (1998). Structured neural networks for pattern recognition. *IEEE Trans. on systems man and cybernetics*, 12(7-8):1173–1180.

[Murphy, 1998] Murphy, R. (1998). *Introduction to AI Robotics*. The MIT Press.

[Murphy, 2000] Murphy, R. (2000). *Introduction to AI robotics (2nd. edition)*. MIT Press.

[Muthuraman, 2005] Muthuraman, S. (2005). *The Evolution of Modular Artificial Neural Networks*. PhD thesis, The Robert Gordon University, Aberdeen, Scotland.

[Muthuraman et al., 2003] Muthuraman, S., MacLeod, C., and Maxwell, G. (2003). The development of modular evolutionary networks for quadrupedal locomotion. In *Proceedings of the 7th IASTED International Conference on Artificial Intelligence and Soft Computing*, pages 268–273.

[Nakada et al., 2004] Nakada, K., Asai, T., and Y., A. (2004). Biologically-inspired locomotion controller for a quadruped walking robot: analog ic implementation of a cpg-based controller. *Journal of robotics and mechatronics*, 16(4):398–403.

[Nelson et al., 2003a] Nelson, A., Grant, E., Barlow, G., and Henderson, T. (2003a). A colony of robots using vision sensing and evolved neural controllers. In *Proceedings of the Intelligent Robots and Systems*.

[Nelson et al., 2003b] Nelson, A., Grant, E., Barlow, G., and White, M. (2003b). Evolution of autonomous robot behaviors using relative competitive fitness. In *Proceedings of the IEEE International Conference, Integration of Knowledge Intensive Multi-Agent Systems: KIMAS'03: Modeling, Exploration, and Engineering Systems*.

[Nelson et al., 2002] Nelson, A., Grant, E., and Lee, G. (2002). Using genetic algorithms to capture behavioral traits exhibited by knowledge based robot agents. In *Proceedings of the ISCA 15th International Conference: Computer Applications in Industry and Engineering*.

[Nolfi, 1997] Nolfi, S. (1997). Using emergent modularity to develop control systems for mobile robots. *Adaptative Behavior*, 5(3-4):343:364.

[Nolfi and Floreano, 1998] Nolfi, S. and Floreano, D. (1998). Coevolving predator and prey robots: Do "arms races" arise in artificial evolution? *Artificial Life*, 4(4):311–335.

[Nolfi and Floreano, 2000] Nolfi, S. and Floreano, D. (2000). *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. The MIT Press.

[Nolfi and Marocco, 2002] Nolfi, S. and Marocco, D. (2002). Active perception: A sensorimotor account of object categorization. In *Proceedings of the VII International Conference on Simulation of Adaptive Behavior*.

[Nolfi and Tani, 1999] Nolfi, S. and Tani, J. (1999). Extracting regularities in space and time through a cascade of prediction networks: The case of a mobile robot navigating in a structured environment. *Connection Science*, 11(2):125–148.

[Nolfi S., 2000] Nolfi S., M. D. (2000). Evolving visually-guided robots able to discriminate between different landmarks. In *From Animals to Animats 6. Proceedings of the VI International Conference on Simulation of Adaptive Behavior*.

[Nordin et al., 1998] Nordin, P., Banzhaf, W., and Brameier, M. (1998). Evolution of a world model for a miniature robot using genetic programming. *Robotics and Autonomous Systems*, 25:10–16.

[O'Hare and Jennings, 1996] O'Hare, G. M. P. and Jennings, N. R., editors (1996). *Foundations of Distributed Artificial Intelligence*. John Wiley.

[Olmer et al., 1996] Olmer, M., Nordin, P., and Banzhaf, W. (1996). Evolving real-time behavioral modules for a robot with gp. In *Proceedings of the 6th International Symposium on Robotics and Manufacturing*.

[Ostergaard and Lund, 2003] Ostergaard, E. H. and Lund, H. H. (2003). Co-evolving complex robot behavior. In *Proceedings of ICES'03, The 5th International Conference on Evolvable Systems: From Biology to Hardware*.

[Paine and Tani, 2004] Paine, R. W. and Tani, J. (2004). Motor primitive and sequence self-organization in a hierarchical recurrent neural network. *Neural Networks*, 17:1291–1309.

[Paine and Tani, 2005] Paine, R. W. and Tani, J. (2005). How hierarchical control self-organizes in artificial adaptive systems. *Adaptive Behavior*, 13(3):211–225.

[Pardoe et al., 2005] Pardoe, D., Ryoo, M., and Miikkulainen, R. (2005). Evolving neural network ensembles for control problems. In *Proceedings of the GECCO'05*.

[Parisi, 2004] Parisi, D. (2004). Internal robotics. *Connection science*, 16(4):325–338.

[Pasemann, 1998] Pasemann, F. (1998). Evolving neurocontrollers for balancing an inverted pendulum. *Network: Computation in Neural Systems*, 9:495–511.

[Pearson and Gordon, 2000] Pearson, K. and Gordon, J. (2000). *Principles of Neural Science*, chapter Spinal reflexes. McGraw-Hill.

[Pendrith, 1994] Pendrith, M. (1994). On reinforcement learning of control actions in noisy and non-markovian domains. Technical Report UNSW-CSE-TR-9410, School of computer science and engineering, The University of New South Wales, Sydney.

[Pfeifer and Bongard, 2007] Pfeifer, R. and Bongard, J. (2007). *How the body shapes the way we think*. The MIT Press.

[Pfeifer and Scheier, 1997] Pfeifer, R. and Scheier, C. (1997). Sensory-motor coordination: the metaphor and beyond. *Robotics and Autonomous Systems. Special Issue on "Practice and future of autonomous agents"*, 20:157–178.

[Pfeifer and Scheier, 1999] Pfeifer, R. and Scheier, C. (1999). *Understanding intelligence*. The MIT Press.

[Philipona et al., 2003] Philipona, D., O'Regan, J. K., and Nadal, J.-P. (2003). Is there something out there ? inferring space from sensorimotor dependencies. *Neural Computation*, 15(9):2029–2049.

[Pinker, 1997] Pinker, S. (1997). *How the mind works*. Penguin Press.

[Potter and Jong, 2000] Potter, M. and Jong, K. (2000). Cooperative coevolution: an architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1)(8):1–29.

[R. Calabretta and Wagner, 1998] R. Calabretta, S. Nolfi, D. P. and Wagner, G. P. (1998). Emergence of functional modularity in robots. In R. Pfeifer, B. Blumberg, J.-A. M. and Wilson, S., editors, *Proceedings of From Animals to Animats 5*, pages 497–504. MIT Press.

[Reeve, 1999] Reeve, R. (1999). *Generating walking behaviours in legged robots*. PhD thesis, University of Edinburgh.

[Reeve and Hallam, 2005] Reeve, R. and Hallam, J. (2005). An analysis of neural models for walking control. *IEEE Transactions on Neural Networks*, 16(3):733–742.

[Reisinger, 2003] Reisinger, J. (2003). An overview of modularity in artificial evolutionary systems. Unjobs. Symbiogenesis.

[Reisinger et al., 2004] Reisinger, J., Stanley, K., and Miikkulainen, R. (2004). Evolving reusable neural modules. In *Proceedings of the genetic and evolutionary computation conference (GECCO'04)*.

[Reynolds, 1994] Reynolds, C. (1994). Competition, co-evolution and the game of tag. In *Proceedings of the 4th workshop on artificial life*.

[Righetti and Ijspeert, 2008] Righetti, L. and Ijspeert, A. (2008). Pattern generators with sensory feedback for the control of quadruped locomotion. In *Proceedings of the International Conference on Robotics and Automation*, pages 819–824.

[Rogova, 1994] Rogova, G. (1994). Combining results of several neural netwrok classifiers. 7:771–781, 1994. *Neural networks*, 7:771–781.

[Rueckl et al., 1989] Rueckl, J., Cave, K., and Kosslyn, S. (1989). Why are "what" and "where" processed by separate cortical visual systems?. a computational investigation. *Journal of Cognitive Neuroscience*, 1:171–186.

[Russell and Norvig, 2003] Russell, S. and Norvig, P. (2003). *Artificial Intelligence: a Modern Approach*. Prentice Hall.

[Salzmann, 2003] Salzmann, M. (2003). Développement de contrôleurs pour la locomotion d'un robot quadrupède à base d'oscillateurs non linéaires. Master's thesis, École Polytechnique Fédérale de Lausanne.

[Schmidhuber and Prelinger, 1993] Schmidhuber, J. and Prelinger, D. (1993). Discovering predictable classifications. *Neural computation*, 5(4):625–635.

[Schrott and G., 1995] Schrott and G. (1995). A multi-agent distributed real-time system for a microprocessor field-bus network. In *Proc. of 7th Euromicro Workshop on Real-Time Systems*, pages S. 302–307, Odense, Denmark. IEEE Computer Society Press.

[Seys and Beer, 2004] Seys, C. W. and Beer, R. D. (2004). Evolving walking: the anatomy of an evolutionary search. In *From Animals to Animats: Proceedings of the eighth international conference on simulation of adaptive behavior*, volume 8.

[Sharkey, 1996] Sharkey, A. (1996). On combining artificial neural networks. *Connection science*, 8(3):299–313.

[Sharkey, 1997] Sharkey, A. (1997). Modularity, combining and artificial neural nets. *Connection science*, 9(1):3–10.

[Sharkey and Sharkey, 1997] Sharkey, A. and Sharkey, N. (1997). Combining diverse neural net. *The Knowledge Engineering Review*, 12(3):231–247.

[Silva and Ribeira, 2003] Silva, C. and Ribeira, B. (2003). Navigating mobile robots with a modular neural architecture. *Neural computing & applications*, 12, 3-4:200–211.

[Simon, 1962] Simon, H. (1962). The architecture of complexity. *Proceedings of the American Philosophical Society*, 106:467–482.

[Simon, 1969] Simon, H. (1969). *The sciences of the artificial*. MA. MIT Press, 2007 edition. Spanish edition.

[Sims, 1994] Sims, K. (1994). Evolving virtual creatures. In *Siggraph '94 Proceedings*.

[Sperber, 2002] Sperber, D. (2002). In defense of massive modularity. In *Language, Brain and Cognitive development*. MIT Press.

[Stanley and Miikkulainen, 2002] Stanley, K. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

[Stanley and Miikkulainen, 2003] Stanley, K. O. and Miikkulainen, R. (2003). A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130.

[Steels, 1994] Steels, L. (1994). Emergent functionality in robotic agents through on-line evolution. In Brooks, R. and Maes, P., editors, *Artificial Life IV, Proc. of the Fourth Int. Workshop on the Synthesis and Simulation of Living Systems*, pages 8—-16.

[Stone, 2000] Stone, P. (2000). *Layered learning in multiagent systems*. Bradford.

[Stone and Veloso, 2000] Stone, P. and Veloso, M. (2000). Layered learning. In *Proceedings of the 11th European Conference on Machine Learning*, volume 1810, pages 369–381. Springer, Berlin.

[Suganthan, 1999] Suganthan, P. (1999). Hierarchical overlapped som's for pattern classification. *IEEE trans. on neural networks*, 10(1):193–196.

[Sutton and Barto, 1998] Sutton, R. and Barto, A. (1998). *Reinforcement learning: an introduction*. MA: MIT Press.

[Tani and Nolfi, 1999] Tani, J. and Nolfi, S. (1999). Learning to perceive the world as articulated: an approach for hierarchical learning in sensory-motor systems. *Neural Networks*, 12:1131–1141.

[Thangavelautham and D'Eleuterio, 2004] Thangavelautham, J. and D'Eleuterio, G. (2004). Learning from insects: development of an emergent task decomposition network for collective robotics. In *6th Dynamics and control of systems and structures in space conference*.

[Tsai et al., 1994] Tsai, W., Tai, H., and Reynolds, A. (1994). An art2-bp supervised neural net. In *Proc. World Congress on Neural networks*.

[Téllez and Angulo, 2004] Téllez, R. and Angulo, C. (2004). Evolving cooperation of simple agents for the control of an autonomous robot. In *Proceedings of the 5th IFAC Symposium on Intelligent Autonomous Vehicles*.

[Téllez and Angulo, 2007] Téllez, R. and Angulo, C. (2007). Webots software review. *Artificial Life*, 13(3):313–318.

[Téllez and Angulo, 2008] Téllez, R. and Angulo, C. (2008). *Encyclopedia of Artificial Intelligence*, chapter Modularity in artificial neural networks. Information Science Reference.

[Téllez et al., 2006] Téllez, R., Angulo, C., and Pardo, D. (2006). Evolving the walking behaviour of a 12 dof quadruped using a distributed neural architecture. In *Proceedings of the 2nd International Workshop on Biologically Inspired Approaches to Advanced Information Technology*.

[Urzelai et al., 1998] Urzelai, J., Floreano, D., Dorigo, M., and Colombetti, M. (1998). Incremental robot shaping. *Connection Science*, 10:341–360.

[Valsalam and Miikkulainen, 2008] Valsalam, V. and Miikkulainen, R. (2008). Modular neuroevolution for multilegged locomotion. In *Proceeedings of the Genetic and Evolutionary Computation Conference*.

[Valsalam and Miikkulainen, 2009] Valsalam, V. K. and Miikkulainen, R. (2009). Evolving symmetric and modular neural networks for distributed control. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

[Vlassis, 2003] Vlassis, N. (2003). A concise introduction to multiagent systems and distributed ai. Technical report, Informatics Institute University of Amsterdam.

[von Foerster, 1970] von Foerster, H. (1970). *Cognition: a multiple view*, chapter Thoughts and notes on cognition, pages 25–48. Spartan Books.

[Wagner, 2000] Wagner, K. (2000). Cooperative strategies and the evolution of communication. *Artificial Life*, 6(2):149–179.

[Walker et al., 2003] Walker, J., Garret, S., and Wilson, M. (2003). Evolving controllers for real robots: a survey of the literature. *Adaptive behavior*, 11(3):179–203.

[Watkins and Dayan, 1992] Watkins, C. and Dayan, P. (1992). Q-learning. *Machine Learning*, 3(8):279–292.

[Watson, 2002a] Watson, R. (2002a). Modular interdependency in complex dynamical systems. In *Proceedings of the 8th International Conference on the Simulation and Synthesis of Living Systems*.

[Watson, 2002b] Watson, R. A. (2002b). *Compositional evolution: interdisciplinary investigations in evolvability, modularity and symbiosis*. PhD thesis, The Faculty of the Graduate School of Arts and Sciences of Brandeis University.

[Watson et al., 1998] Watson, R. A., Hornby, G. S., and Pollack, J. B. (1998). Modeling building-block interdependency. In Koza, J. R., editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA. Stanford University Bookstore.

[Weiskopf, 2002] Weiskopf, D. A. (2002). A critical review of jerry a. fodor's the mind doesn't work that way. *Philosophical psychology*, 15(4):551–562.

[Werner and Dyer, 1992] Werner, G. and Dyer, M. (1992). Evolution of communication in artificial organisms. In Langton, C., Taylor, C., Farmer, D., and Rasmussen, S., editors, *Artificial Life II*, pages 659–687, Redwood City, CA. Addison-Wesley Pub.

[Whiteson and Stone, 2003] Whiteson, S. and Stone, P. (2003). Concurrent layered learning. In Rosenschein, J. S., Sandholm, T., Wooldridge, M., and Yokoo, M., editors, *Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 193–200, New York, NY. ACM Press.

[Whitley et al., 1993] Whitley, D., Dominic, S., Das, R., and Anderson, C. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13(2-3):259–284.

[Whitley et al., 1995] Whitley, D., Gruau, F., and Pyeatt, L. (1995). Cellular encoding applied to neurocontrol. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 460–469. Morgan Kaufmann.

[Whitley et al., 1991] Whitley, D., Mathias, K., and Fitzhon, P. (1991). Delta-coding: an iterative search strategy for genetic algorithms. In *Proceedings of the fourth international conference on genetic algorithms*.

[Wierstra et al., 2005] Wierstra, D., Gomez, F., and Schmidhuber, J. (2005). Modeling systems with internal state using evolino. In *Proc. of the 2005 conference on genetic and evolutionary computation (GECCO)*, pages 1795–1802.

[Wischmann et al., 2005] Wischmann, S., Hülse, M., and Pasemann, F. (2005). (co)evolution of (de)centralized neural control for a gravitationally driven machine. In *Proceedings of the ECAL2005*.

[Xu et al., 1995] Xu, L., Jordan, M. I., and Hinton, G. E. (1995). An alternative model for mixtures of experts. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems*, volume 7, pages 633–640. The MIT Press.

[Yamaguchi and Itakura, 1999] Yamaguchi, S. and Itakura, H. (1999). A modular neural network for control of mobile robots. In *Proceedings of the 6th International Conference on Neural Information Processing*.

[Yamauchi and Beer, 1994] Yamauchi, B. and Beer, R. (1994). Integrating reactive, sequential, and learning behavior using dynamical neural networks. In *SAB94: Proceedings of the third international conference on Simulation of adaptive behavior : from animals to animats 3*, pages 382–391, Cambridge, MA, USA. MIT Press.

[Yao and Liu, 1996] Yao, X. and Liu, Y. (1996). Ensemble structure of evolutionary artificial neural networks. In *Proc. of the 1996 IEEE Int'l Conf. on Evolutionary Computation*.

[Yong and Miikkulainen, 2001] Yong, H. and Miikkulainen, R. (2001). Cooperative coevolution of multiagent systems. Technical Report AI01-287, Department of computer sciences, University of Texas.

[Ziemke and Bodén, 1999] Ziemke, T. Carlsson, J. and Bodén, M. (1999). An experimental comparison of weight evolution in neural control architectures for a 'garbage-collecting' khepera robot. In Löffler, A. Mondada, F. and Rückert, U., editors, *Experiments with the Mini-Robot Khepera - Proceedings of the 1st International Khepera Workshop*, pages 31–40.

[Ziemke, 2005] Ziemke, T. (2005). Cybernetics and embodied cognition: On the construction of realities in organisms and robots. *Kybernetes*, 34(1/2):118–128.

A

Simulator analysis

A.1 Aibo robot

The Aibo robot is a quadruped robot with dog shape developed by Sony. Aibo has four legs with three degrees of freedom each. It can walk, recognize people and voice and maintain its own agenda.

Aibo has sensor positions at each joint. It also has touch sensors on the every foot that alow it to detect when it is touching the ground. Additionally, it has a three axes accelerometer that allows it to detects its position.

What is interesting about this robot are two things:

first, it has a high complexity level, in terms of number of sensors and actuators. Most of the behaviors evolved required the coordination of more than 30 devices.

Second, it can be programmed by the user, even if using a complex API.

Because of all that, it is an interesting platform where to test the DAIR architecture. However, performing test directly into the Aibo robot is very time consuming, and dangerous for the robot itself. To avoid those problems, a simulator of the robot is preferred. Hence, experiments are performed in the simulator, and once the results obtained are satisfactory, the results are transfered to the real robot and tested on it.

A.2 Aibo simulator

To achieve a correct transfer from the simulator generated controller to the real robot controller, the simulator must be accurate enough to guarantee that the results obtained on it will be similar in the real robot. So, the selection of the simulator has a great importance for the work performed in this thesis.

At the moment of starting this thesis, a simulation of the Aibo robot model ERS-210 existed within the Webots simulator. However, the Aibo model used in this thesis is the ERS-7. So, we modified the Webots simulator to include the new ERS-7 simulation[1]. Those modifications can be found at current versions

[1] An initial 3D model of the ERS-7 was already made by Olivier Michel. Starting from that rough model, we modified it, and adapted the control panel, and transference system, taking

of the commercial simulator.

In order to tune the 3D model and tranference system for the ERS-7, we performed a series of tests that characterize the simulation in all posible ways that the robot can encounter. Next section describes the tests performed and the results obtained.

A.3 Evaluation of the simulation acuracy

In this section we qualitatively and quantitatively evaluate the Aibo simulation in Webots by comparing the simulation with the real Aibo while performing various tasks. We implement measurements on the robot in static and dynamic situations, and we compare performance between them. The comparison procedure is always the same: a Webots controller is created for the simulation. This controller makes the robot perform an action that we want to measure. The controller is then executed in the simulator and the measure taken. Next, the controller is cross-compiled using Webots cross-compilation system, and executed on the real robot. Finally, the measure of the real robot is taken and compared with the simulation result. The cross-compilation of the code ensures that the same control program will be executed in both simulator and real robot, and allows us to perform a fair comparison of performance between them.

A.3.1 Static measurement

The first comparison consists of measuring how the simulation differs from the real robot when confronted to extreme static positions of the joints. In this case, a Webots controller is created which slowly moves one of the leg joints to the maximum or minimum of the joint range, starting from an initial natural position. The controller is executed in the simulator and in the real robot, but in both cases, only one joint is moved at the same time. Final positions reached by simulator and real robot are recorded and compared visually and numerically. The velocity at which joints are moved is very slow, so static states can be assumed at any position of the joint.

Due to the fact that a single joint was moved at the same time, the robot was required to slide over the ground. For this reason, the friction against the ground was reduced to the minimum. In simulation this was achieved by reducing the friction coefficient of the simulated ground. In real robot this was achieved by placing some small sheets of paper under the paws of the robot.

The results of those tests can be seen in figure A.1. We observe that both simulator and real robot achieve the same final position and, morever, the trajectory described by both from the initial position to the final one, is practically identical.

as point of departure the previous system developed for the ERS-210 by Lukas Holh.

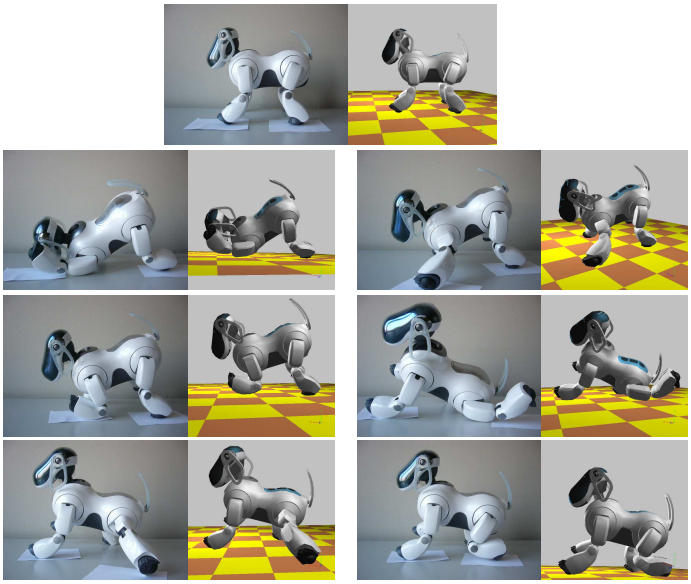


Figure A.1: List of pairs of static figures obtained in the simulator and the real robot when moving a joint at a time. The first two pictures show the initial position, and the resting pairs are the resultings of moving joints LeftFront J1, J2, J3, and Left Hind J1, J2 and J3 respectively.

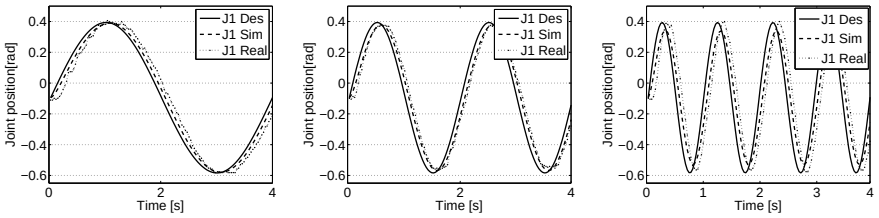


Figure A.2: Trajectories obtained for the J1 joint at three different frequencies. From left to right, oscillations at 0.25 Hz, 0.5 Hz and 1 Hz

A.3.2 Dynamic measurement

We carried out dynamic measurements, to analyze the difference between the simulation and the real robot when they perform a movement. We will show the results obtained from three different experiments, with an increasing degree of complexity in the robot behavior. Experiments conducted consisted of the characterization of the movement of one leg alone, the characterization of the whole robot when performing a walking gait, and the characterization of the whole robot when performing a walking gait and stopping at a specified distance to an obstacle detected by the long distance sensor.

Movement of one leg

In this experiment, the three joints of a leg are moved following a sinusoid trajectory. Trajectories obtained from the real and the simulated robot were obtained and a measure of error calculated, comparing differences between the desired trajectories and the ones obtained in both cases. For every joint three trajectories where performed at three different frequencies of oscillation. The error is calculated as the mean square error per period, normalised by the amplitude.

Figure A.2 shows the trajectories obtained for joint J1 at every testing frequency, including the desired trajectory, the trajectory obtained by the simulated leg, and the trajectory of the real robot leg. Figure A.3 summarizes the errors obtained between desired trajectory and actual trajectory, for the simulator and the real robot for all types of joints.

From both figures we observe that the error obtained at these frequencies is small. Nevertheless, error increases exponentially with the frequency of the oscillation, and higher frequencies started to be difficult to follow for the joints, especially for the real robot. At the same time, errors between simulation and real robot also increased exponentially showing that at high frequencies the simulator starts to differ more from the real robot. The two types of errors present at higher frequencies are phase differences and attenuation of the shape trajectory, i.e. typical tracking errors of PD controller. However, these are only observed at frequencies far from the typical use for this robot and should not affect much the simulation.

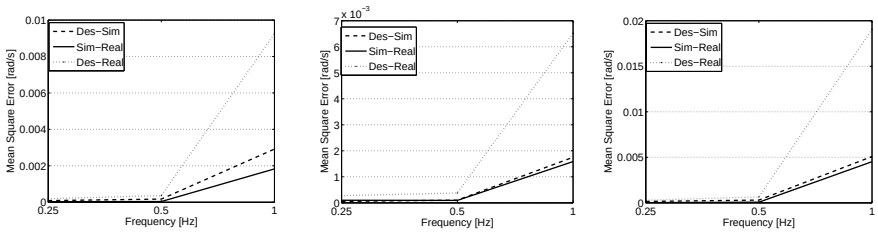


Figure A.3: Error measures obtained for the three types of joints. Each figure shows the error between the desired trajectory (Des), the simulator (Sim) and the real robot (Real). From left to right, errors for joints J1, J2 and J3

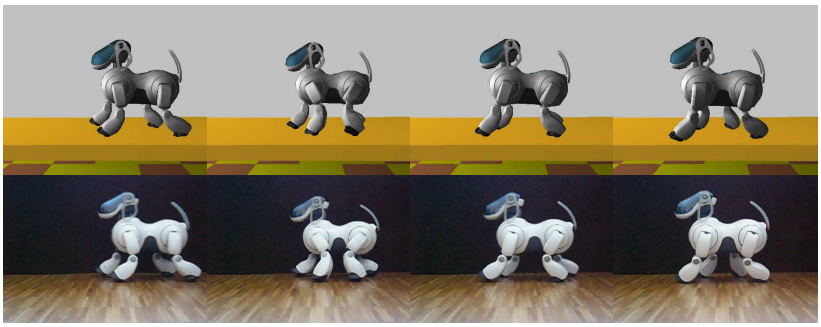


Figure A.4: Walking sequence performed by simulator and real robot

Performing a walking gait

This section shows how similar the simulator and the real robot behave when implementing a complex movement. Both the simulator and the real robot run a Webots controler that executes an MTN file specifying a walking gait. Any MTN file specifies the exact sequence of movements for each robot joint at any time step in order to generate sequence of movements (in this case a walking pattern). The walking gait obtained for the simulator and the real robot are visually compared, joint trajectories are recorded and compared, and speed is measured and compared.

The robot was placed on a parquet floor and its friction parameter introduced on the simulator. The paws of the robot are made of rubber. Typical values for a parquet floor friction against rubber are between 0.55 and 1.36 [ref]. A mean value of 0.7 was selected. The robot executed then a sequence of five walking steps, and the distance and time to accomplish those was measured.

First, a visual comparison of the walking was performed. Figure A.4 shows a sequence of movements obtained in both systems. Walking gait was very similar with no appreciable visual difference.

Second, the motor positions on each legs were also recorded for comparison. Differences between the positions of the left fore leg in simulation and on the

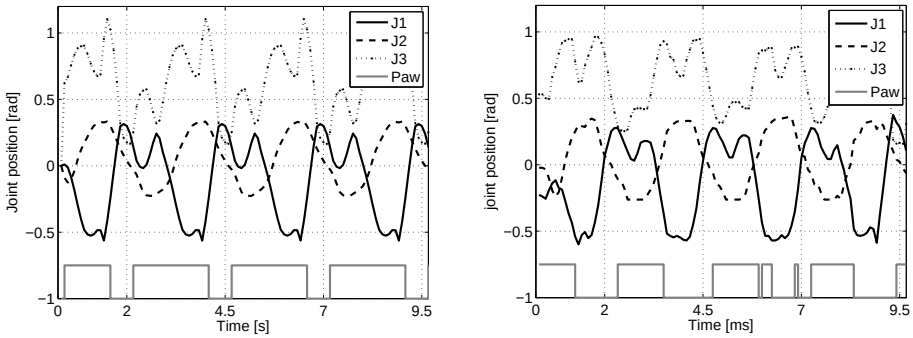


Figure A.5: Joint trajectories during walking for the left fore leg, in the simulator and with the real robot

real robot can be seen in figure A.5. Trajectories are also very similar as the visual inspection of the gait indicated, the only significant difference being the value of the paw sensor. This is due to the fact that real paw is more noisy than the simulated one and some contacts are not detected by it. Figure A.5 only shows joints of one leg, but other legs behave in the same manner (including paw sensor errors).

Finally, velocity in both systems was measured. Due to the chaotic nature of the real robot, a statistical measurement was required to obtain its velocity. A mean value was calculated out of ten runs. For the simulator no statistical measurement was required since no noise was introduced in the simulation and the results could then be repeated all the times with the same final value. Nevertheless, due to small variations in some variables, it was noted that the final distance was not exactly the same in different rounds, but the difference was so small that it was discarded. This difference may indicate the presence of chaotic behavior in the simulator that may affect more complex setups, and may come from the accumulation of numerical errors due to loss of precision.

The measurements of the velocities showed a velocity of 3.50 cm/s for the simulator, and a mean velocity value of 3.25 cm/s for the real robot with a variance of 0.0261.

The results in this section show again that small differences between the simulator and the real robot can be observed, but none of them really significant.

Walking up to a distance

This experiment measures the difference between the simulator and the real robot when the whole robot is put into test, ina behavioral task involving movements and sensor readings.

In this case, Aibo is requested to start walking from an initial position in the same way as in the previous experiment, but to stop after it detects any obstacle at less than 40 cms. Obstacle detection is performed by using the *Far*

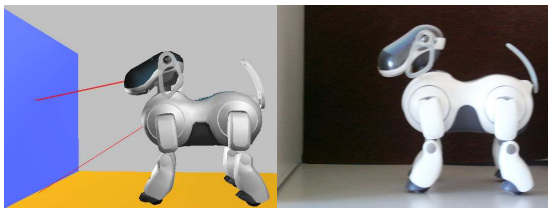


Figure A.6: Simulator and real robot setup for distance measurement

distance sensor situated in the head of the robot. The mechanism is the same as in previous sections: a webots controller is created and executed in both the simulator and the real robot (cross-compiled), and the differences between both measured. In this case, we compare the distance at which the robot stopped from the obstacle, but also all the measures taken by the distance sensor during the walking period. The measures of distance taken during walking are obtained from the far distance sensor of the robot, which directly provides a value of the distance in milimeters.

Since there exist sensor errors in both simulated and real robot (simulated sensors are modeled with noise), a unique and absolute measure of the distance cannot be used for stopping the robot. Because of that, a hysteresis mechanism was implemented, consisting of stopping the robot only after a distance below 40 cm has been detected for ten consecutive time steps of 96 ms. The final distance and the trajectories were measured ten times in each type of robot.

The experiments showed that the simulated robot stopped at a mean distance of 33.6 cm of the obstacle with a variance of 0.22 cm, meanwhile the real robot stopped at a mean distance of 32.72 cm of the obstacle with a variance of 15.67 cm. Eventhough the mean distances at which both robots stopped are very similar, the variance values among them are very different. The simulated robot has a very small variance, only due to the noise of the distance sensor. On the other side, the real robot has a very large variance due to several factors not included in the simulation: first the distance measure method is far from exact in the real robot (it is a manually done). Second, all the joints have their own noise that affect the final position of the robot head, amplifying the difference in the distance measured. Third, the movements on the head of the robot can produce reflections of the infrared ray, producing even more measure differences.

All these differences can be seen in the figure A.7. In that figure, the mean distance measured value obtained at each time step is presented, together with the standard deviation at some randomly selected points. It can be seen that while the simulator presents a *clean* line, the real robot has a more noisy line showing the high variance detected in the measurements.

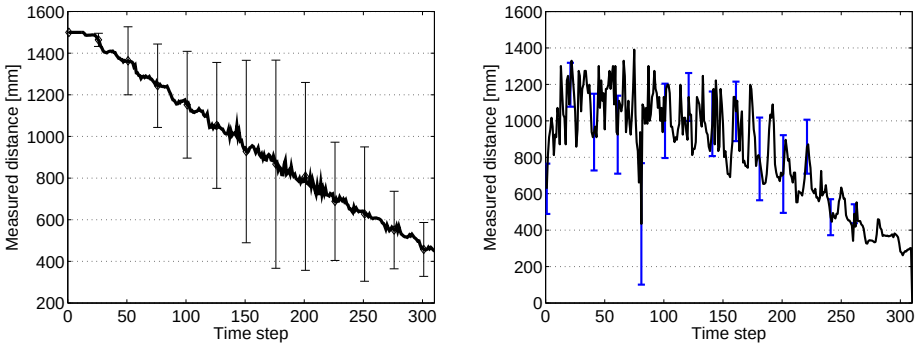


Figure A.7: Mean distance value to an obstacle for the simulator (left) and the real robot (right), obtained during the walking of the robot towards the obstacle. Mean value is represented by thick black line. Vertical lines represent the standard deviation of the measure at those points

B

Connection with simulator

B.1 How to interface the evolutionary algorithm with the simulator for offline evolution

The offline evolutionary proces is performed as follows:

In order to run a simulation, we need to create the simulation environment. This is the 3D representation of the robot, its environment, and all the physical things that are required to participate in the simulation. Then, we have to write two programs which will run at the same time within the simulator: the *supervisor* and the *controller*. The task of the supervisor is to implement the evolutionary process, as it is specified by the given evolutionary algorithm (in our case, the ESP). The controller, instead, executes a given phenotype in the simulator and calculates the reward for its associated genotype. A schematics of how do they interact can be found in figure B.1.

Once the simulation starts, both programs will run at the same time in different threads. The controller will just initialize the robot and remain on a waiting state, waiting for a phenotype to execute on the robot. This phenotype will have to be provided by the supervisor. Since the programs run in different threads, the communication between supervisor and controller is performed through a communication channel specified in the simulation.

The supervisor on its side will start by initializing the genetic algorithm. It initializes its parameters and creates the first random generation of genotypes. Then it goes to the selection and testing of all the genotypes, as was specified in chapter 2. It selects the first genotype to test, converts it to a phenotype (that is, to a neural controller) and sends it to the controller program. At this point, the supervisor will wait until an answer from the controller is produced.

The controller program receives the neural controller and executes it on the robot. This means that the neural controller will receive as inputs the what the robot is sensing on its current situation, and will send to the effectors of the robots the outputs generated by the neural controller. This execution of the neural controller will last for a given amount of time (the time of evaluation) which has been specified before hand. So, during this time, the robot in the simulation will behave on the way indicated by the neural controller. After the

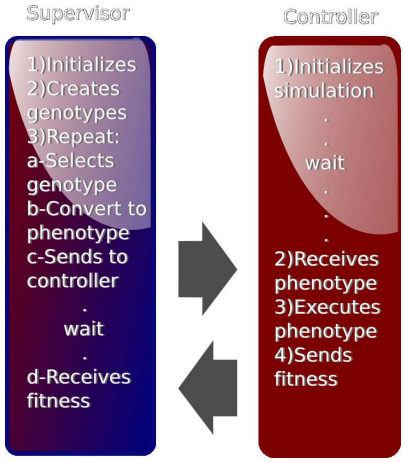


Figure B.1: Schematics of the interface of the evolutionary algorithm with the simulator

time is consumed, the robot will stop, and a measurement of the behavior of the robot will be taken, that is, a fitness measurement which indicates how good or bad the robot behaved. The fitness is then sent to the supervisor as the requested answer.

Once the supervisor receives the fitness for that genotype, it stores that information and selects a new genotype to be tested. The process is then repeated. The genotype is converted to phenotype, and sent to the controller until a response with the fitness is obtained. Once all the genotypes have been tested, they are ordered and a new generation of genotypes is obtained as the evolutionary algorithm specifies. The whole process is then repeated for that new generation. This process will repeat over and over until the number of maximum generations is reached, and the final controller is obtained.

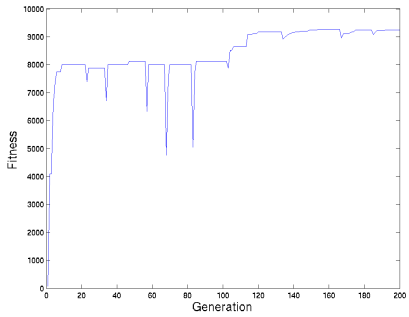
C

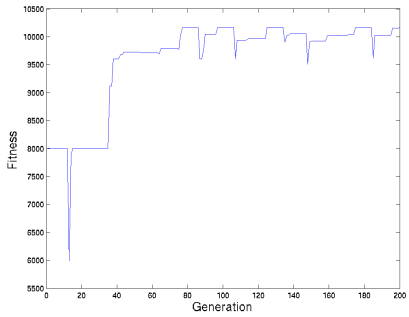
Chapter 4 results

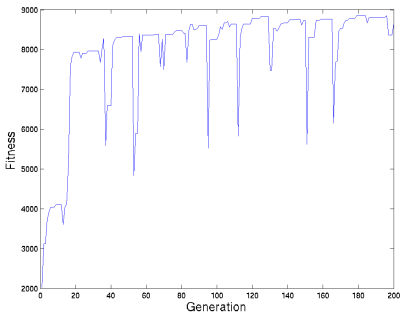
C.1 Experiment results of the contour following behavior

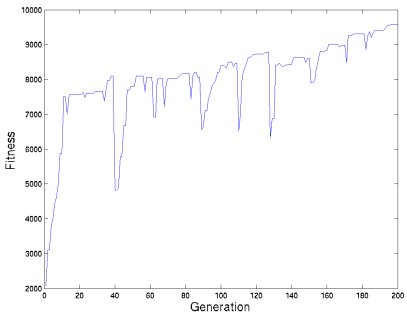
This sections contains the evolution of the fitness for all the seven runs performed for the evolution of each type of controller of section 4.2.6.

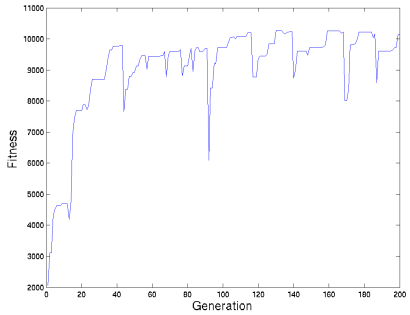
C.1.1 Monolithic controller

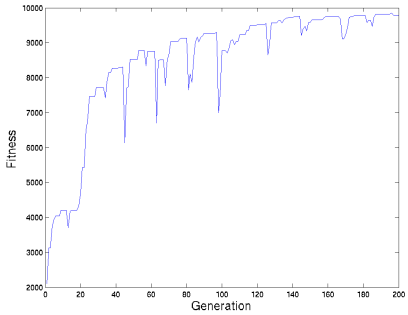


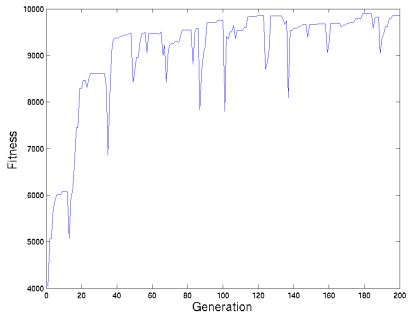




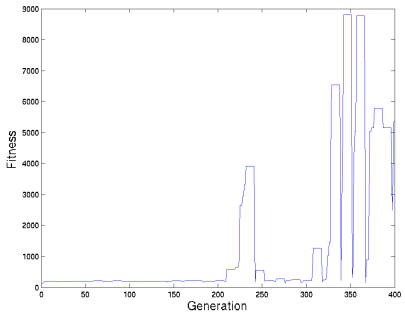


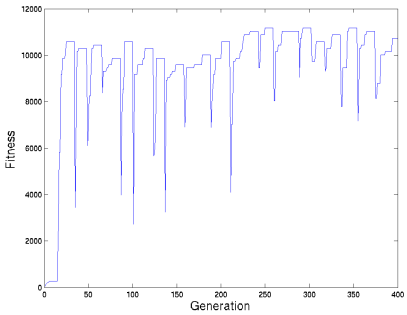


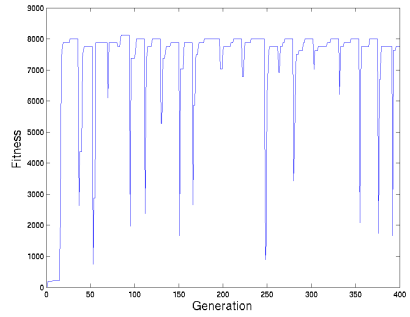


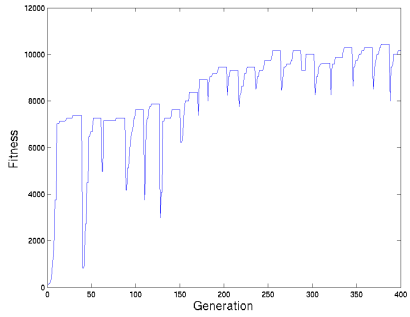


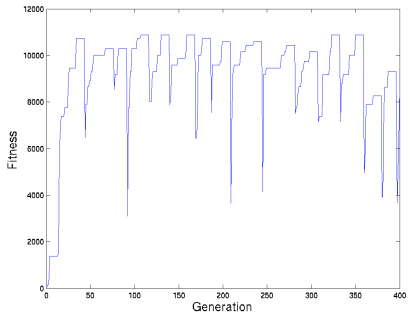
C.1.2 Tactical controller

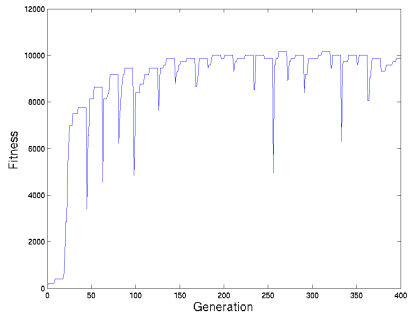


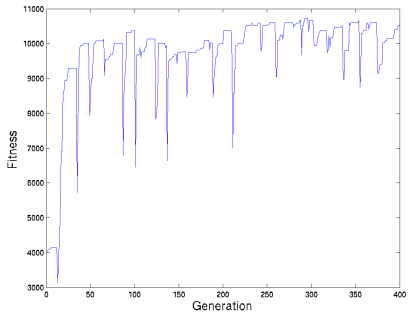



Fitness
Generation

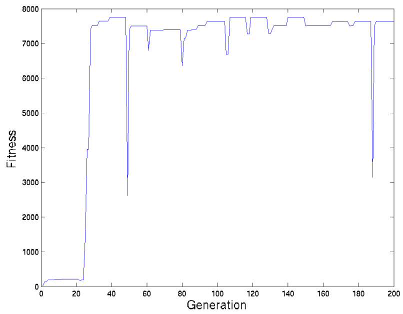

Fitness
Generation

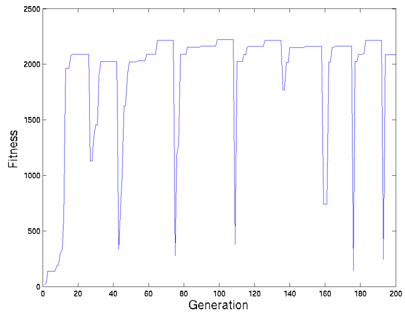

Fitness
Generation

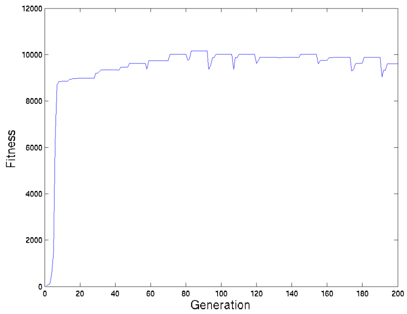

Fitness
Generation

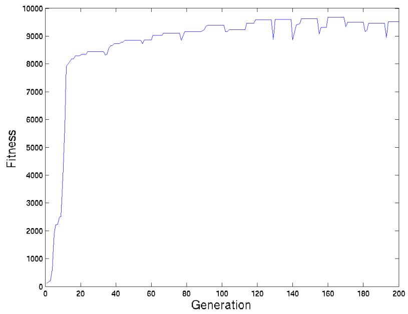

Fitness
Generation

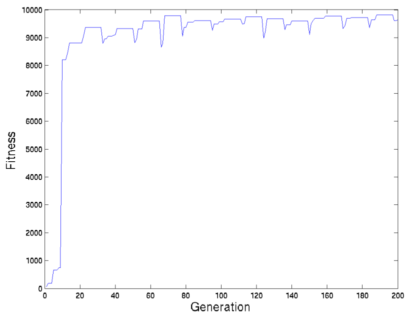
C.1.3 Tactical controller without sensor IHUs

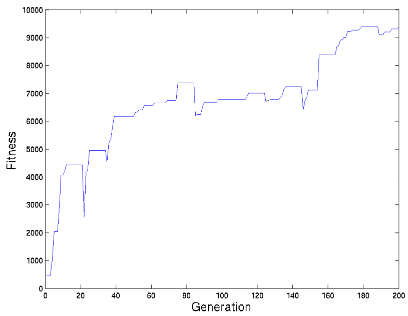


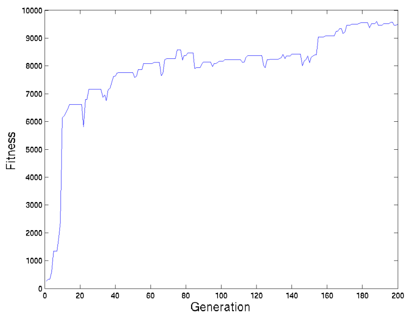






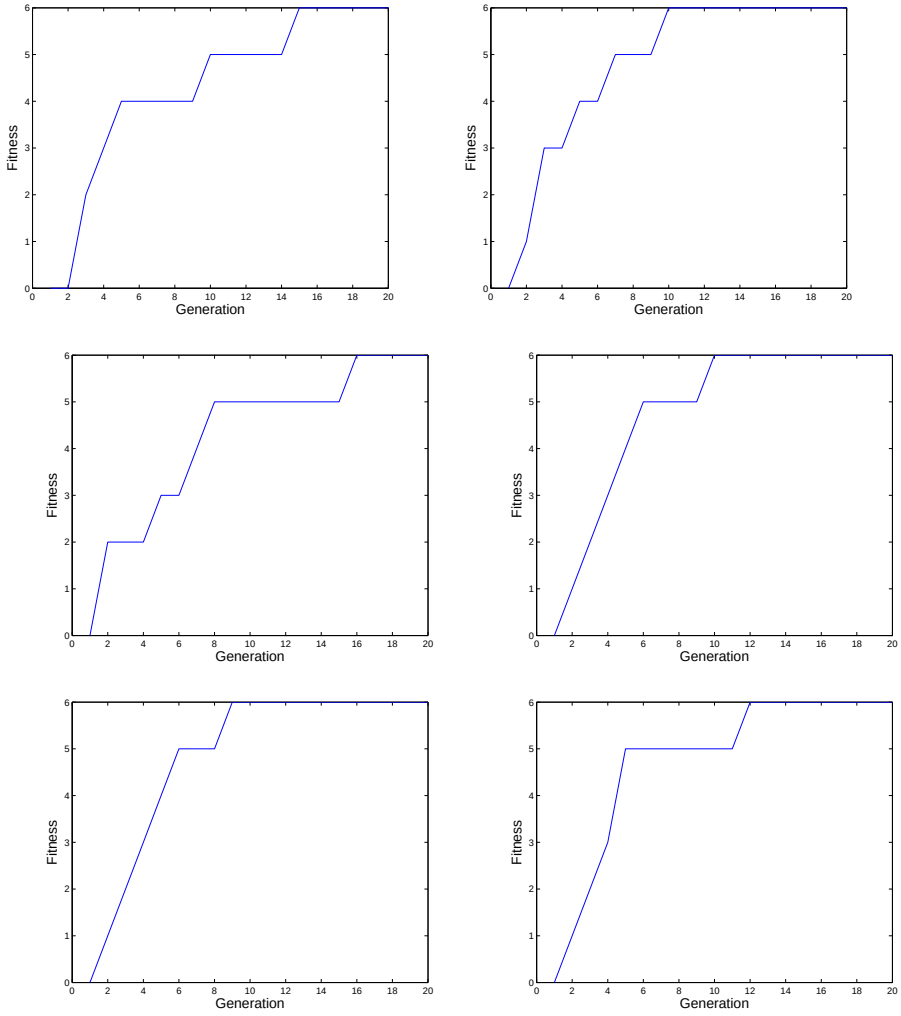


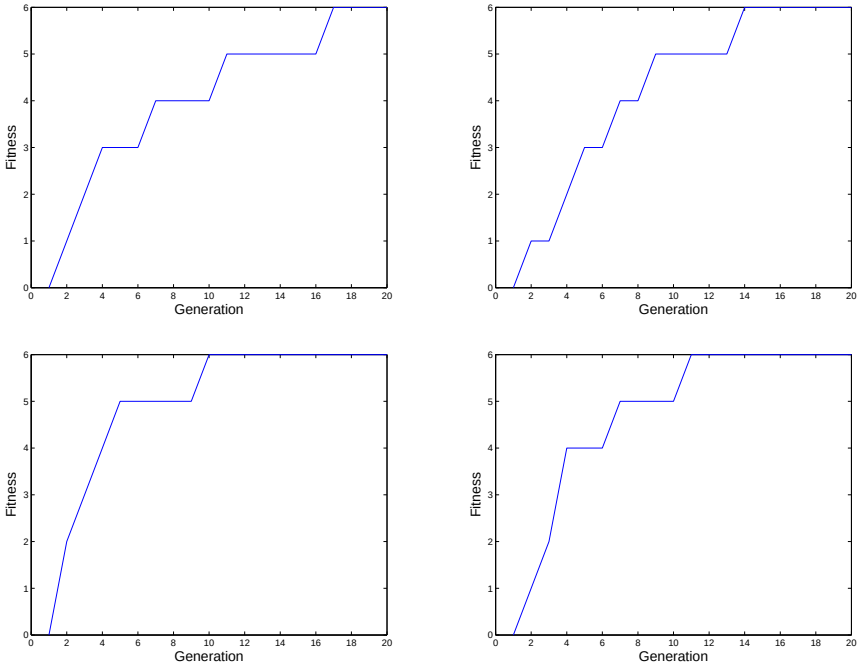




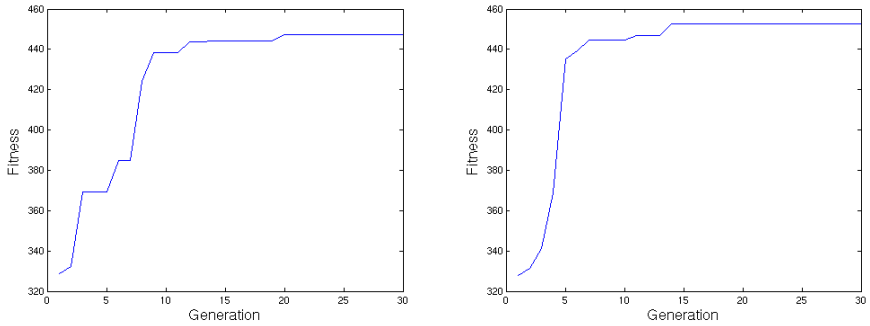
C.2 Experiment results with an Aibo robot

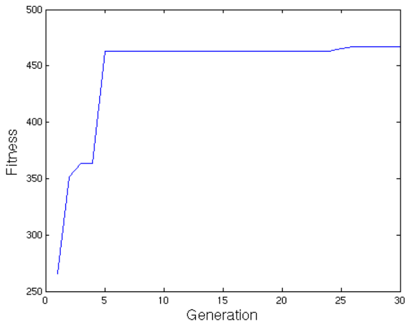
C.2.1 Keep stand up tactical controller

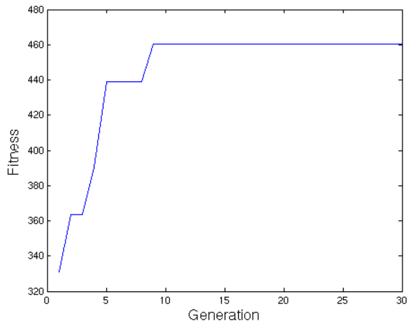


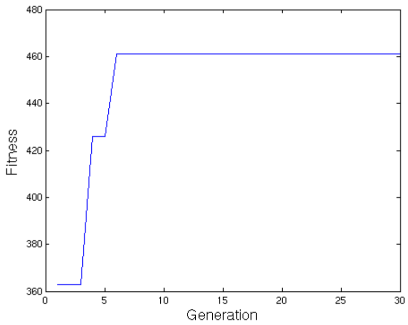


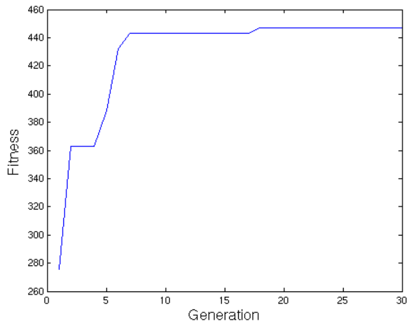
C.2.2 Stand up tactical controller

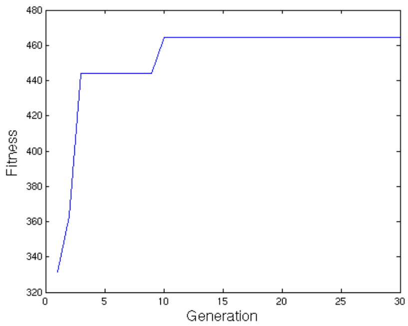


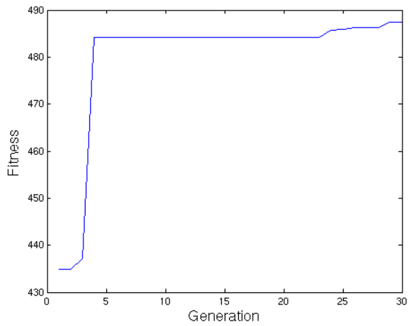

500
450
400
350
300
250
Fitness
0 5 10 15 20 25 30
Generation

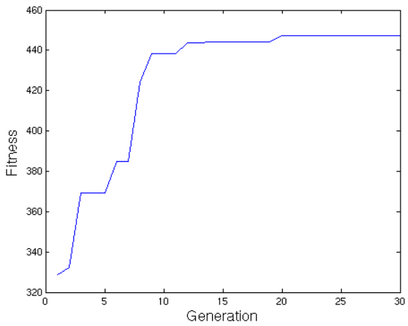

480
460
440
420
400
380
360
340
320
Fitness
0 5 10 15 20 25 30
Generation

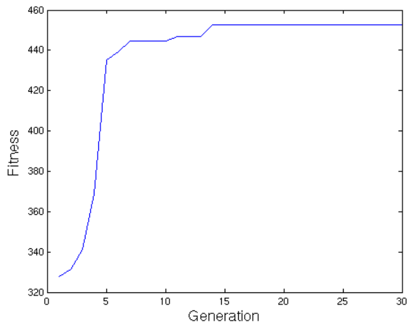

480
460
440
420
400
380
360
Fitness
0 5 10 15 20 25 30
Generation

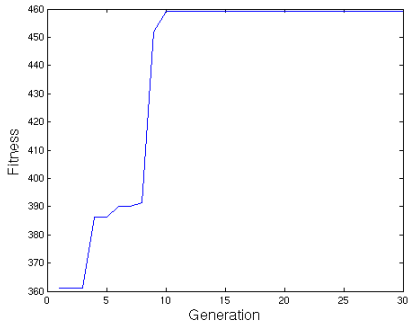

460
440
420
400
380
360
340
320
300
280
260
Fitness
0 5 10 15 20 25 30
Generation

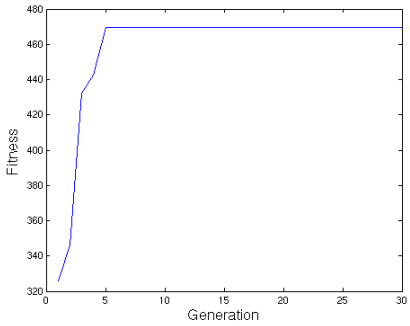

480
460
440
420
400
380
360
340
320
Fitness
0 5 10 15 20 25 30
Generation


490
480
470
460
450
440
430
Fitness
0 5 10 15 20 25 30
Generation


460
440
420
400
380
360
340
320
Fitness
0 5 10 15 20 25 30
Generation


460
440
420
400
380
360
340
320
Fitness
0 5 10 15 20 25 30
Generation


Fitness
Generation
460
450
440
430
420
410
400
390
380
370
360
0
5
10
15
20
25
30


Fitness
Generation
480
460
440
420
400
380
360
340
320
0
5
10
15
20
25
30

D

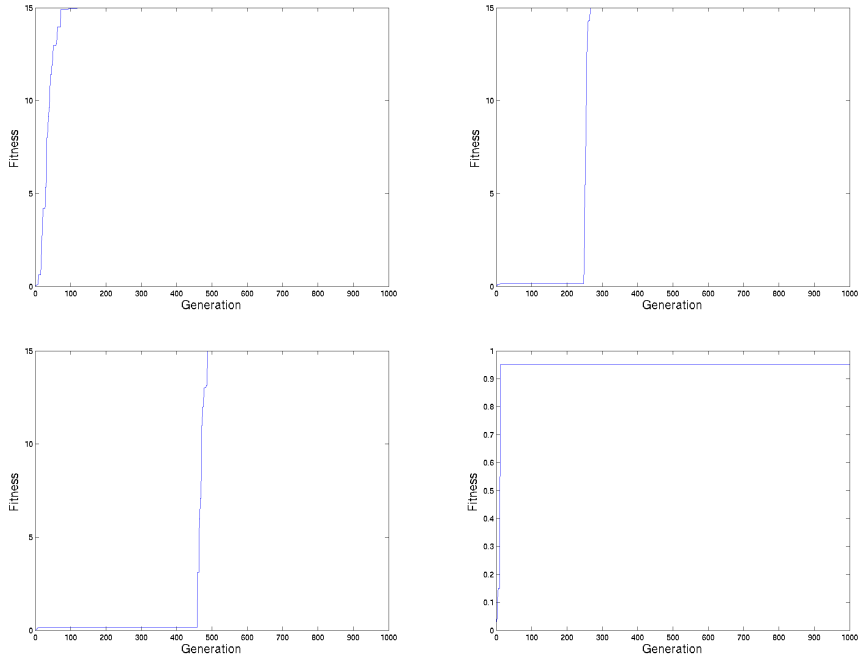
Chapter 5 results

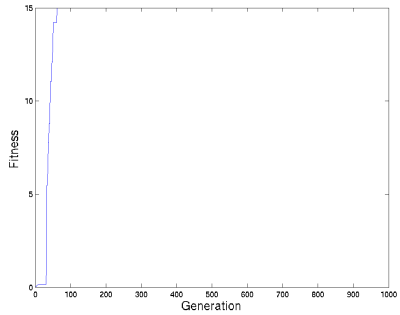
D.1 Experiment results for the different architectures solving the garbage collector

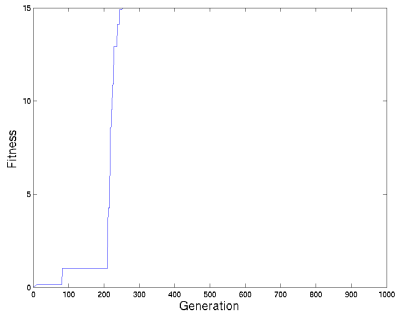
This sections contains the evolution of the fitness for all the ten runs performed for the evolution of each type of controller of chapter 5.

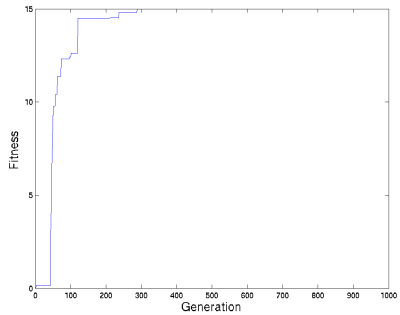
D.1.0.1 Monolithic controller

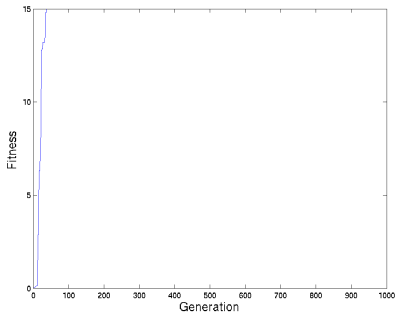
Main evolutionary process

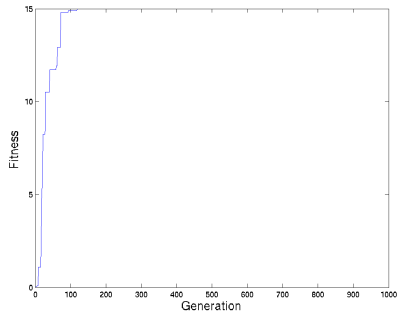


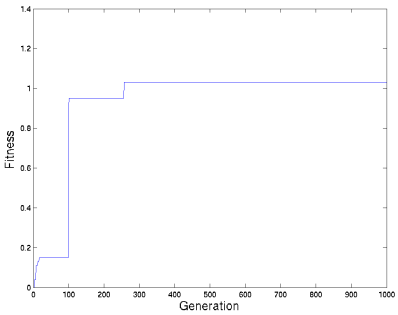




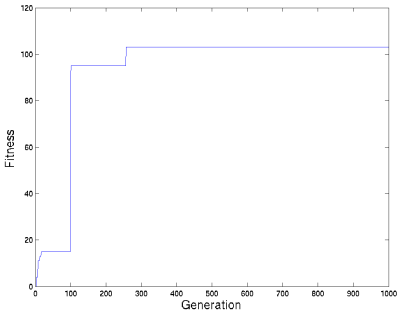


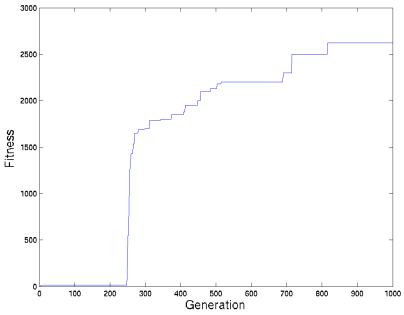


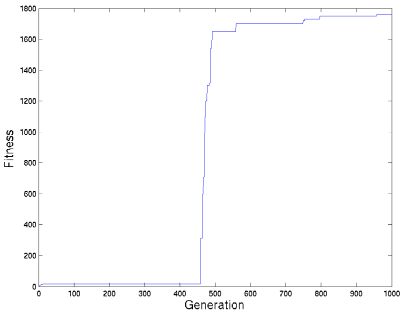


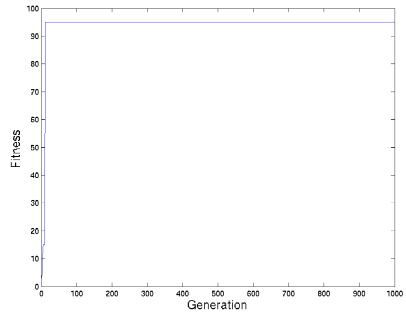


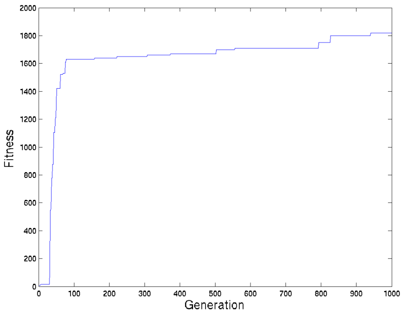
Extra experiment results

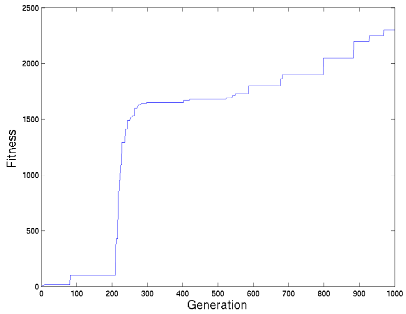


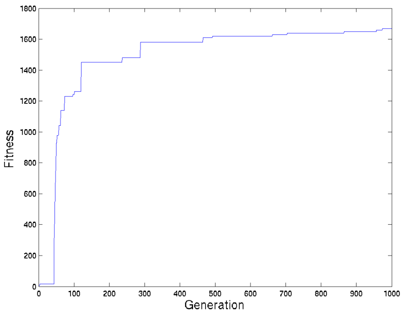


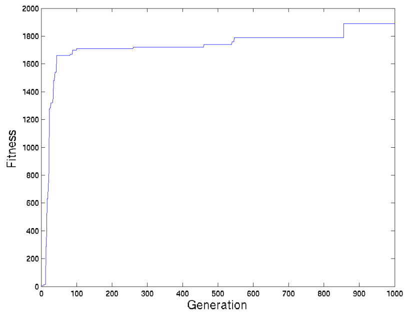

Fitness
Generation

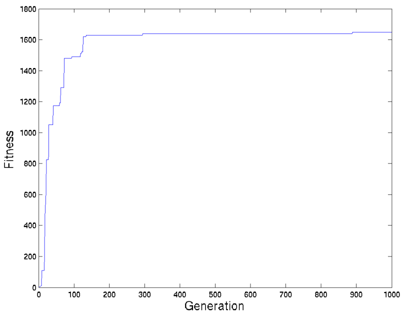

Fitness
Generation

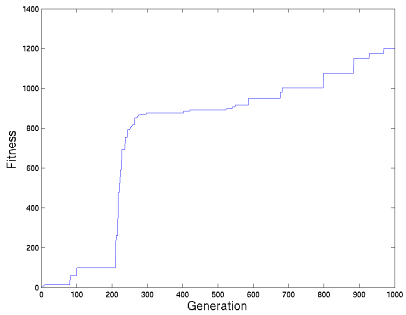

Fitness
Generation


Fitness
Generation


Fitness
Generation

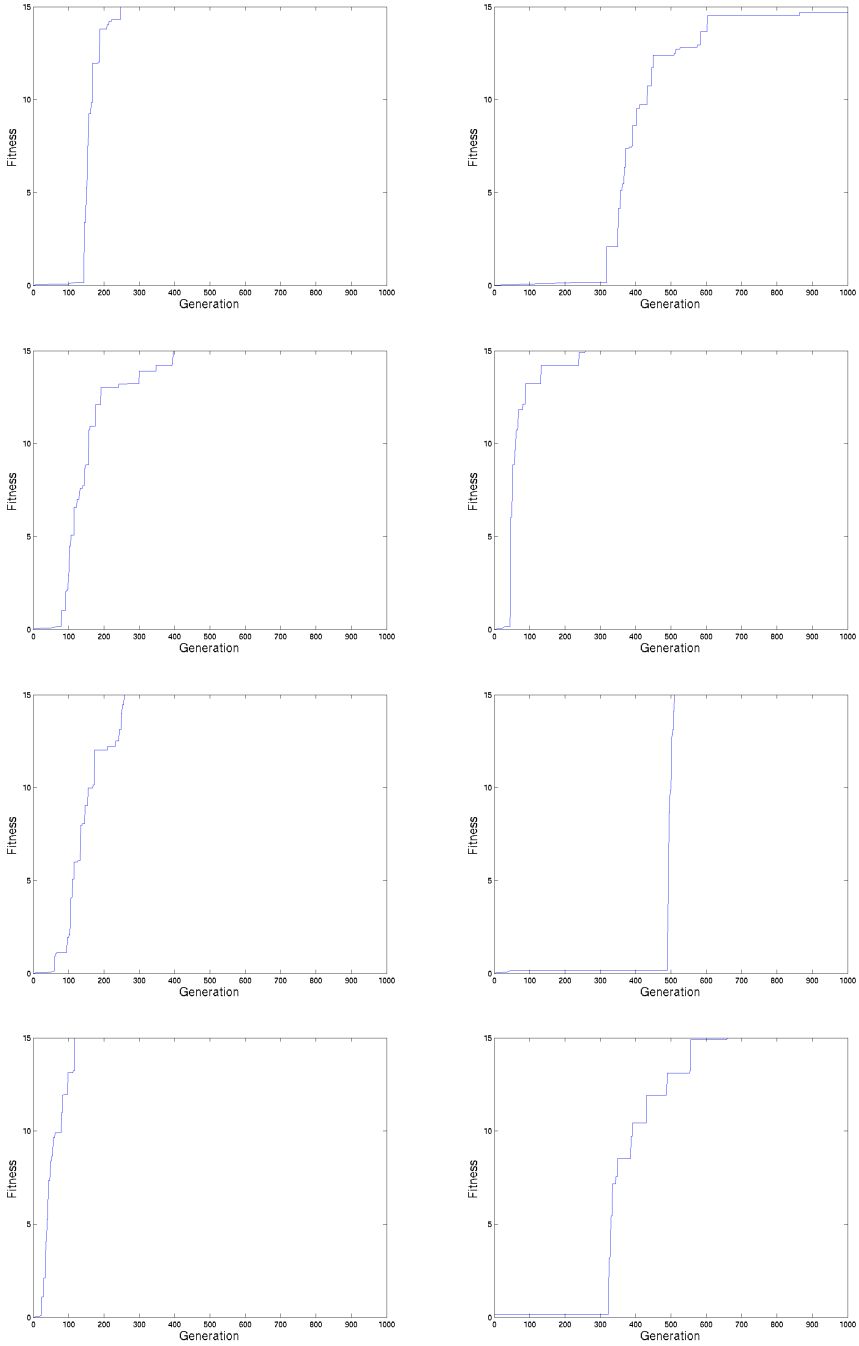

Fitness
Generation

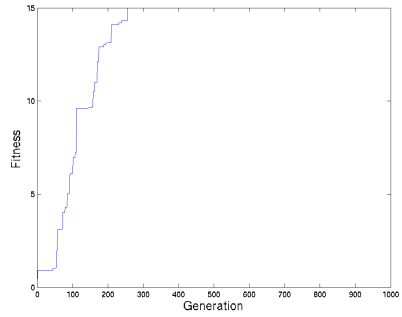

Fitness
Generation

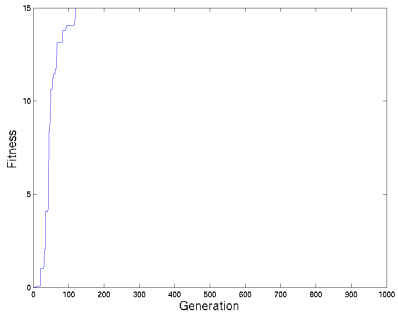

Fitness
Generation

D.1.0.2 Tactical controller

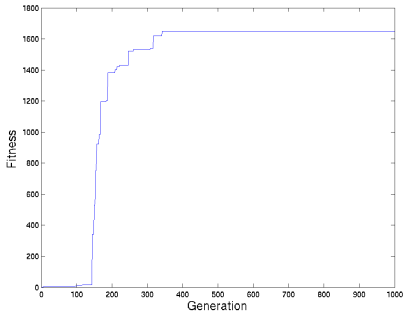
Main evolutionary process

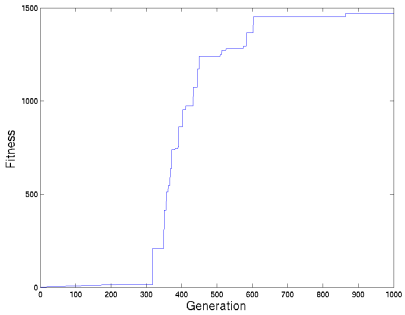


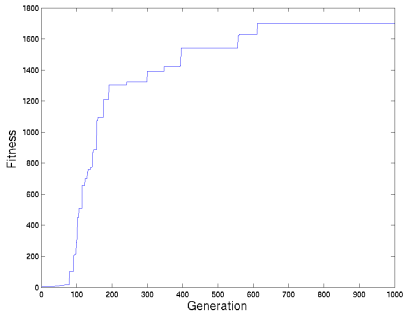


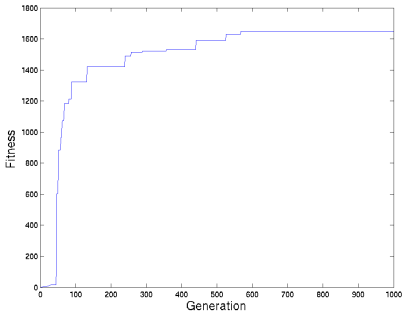


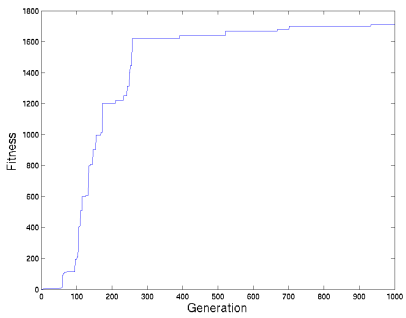
Extra experiment results

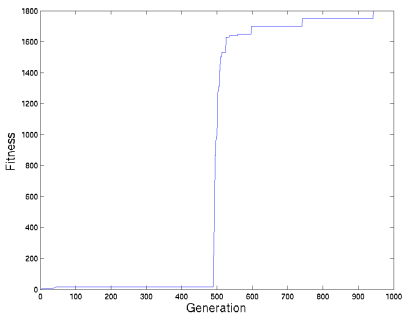


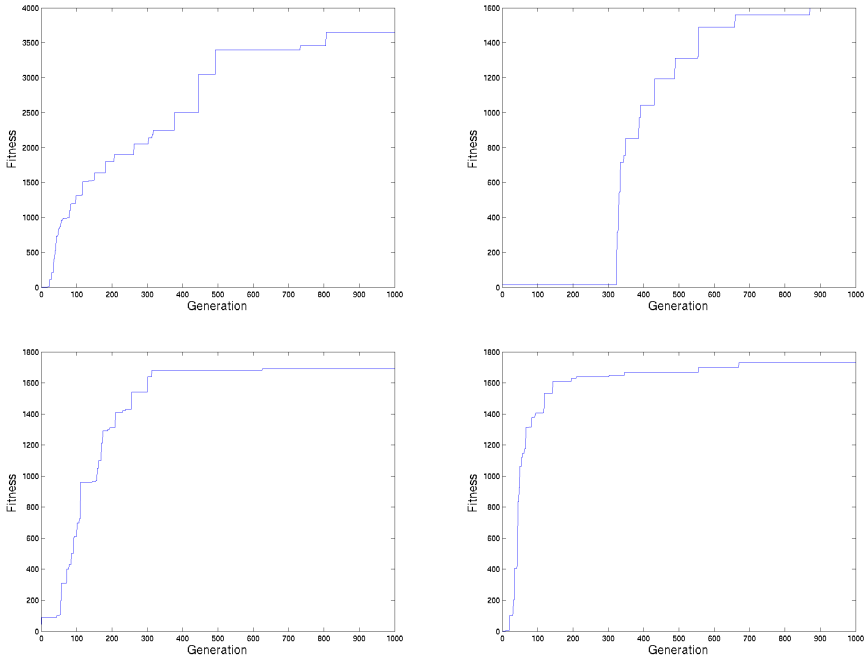






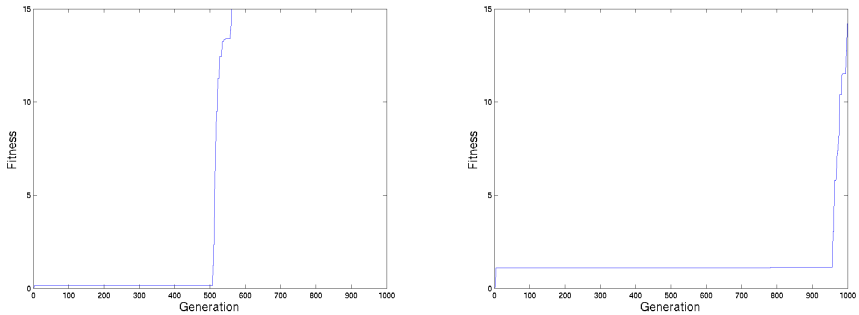


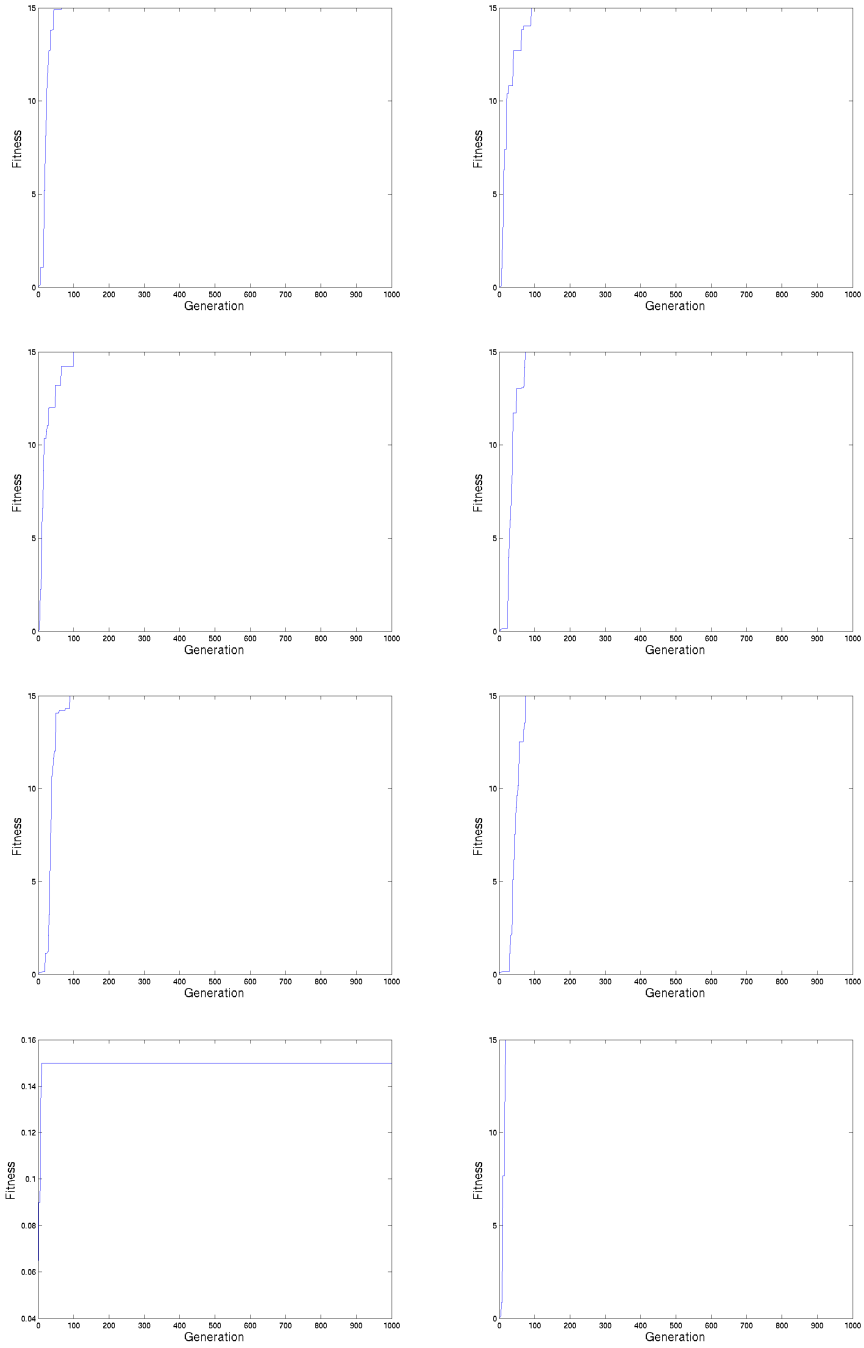




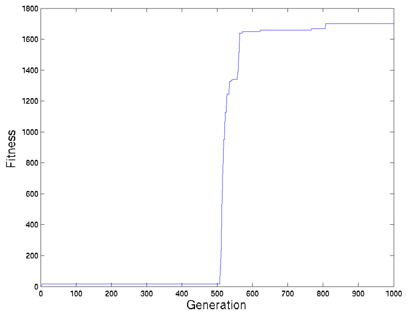
D.1.0.3 Tactical controller without sensor IHUs

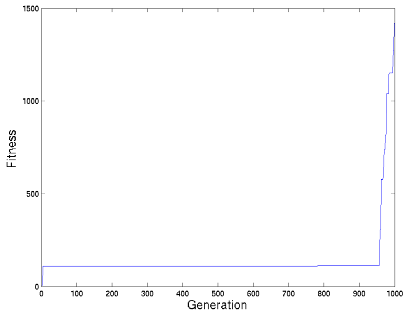
Main evolutionary process

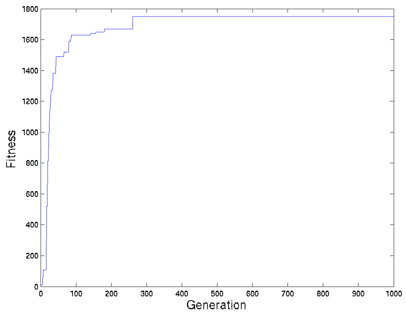


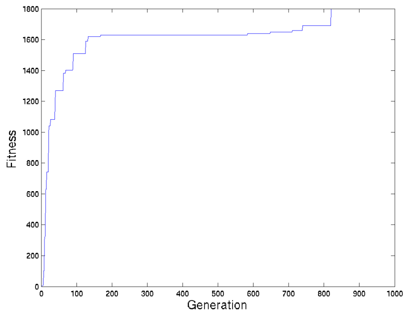


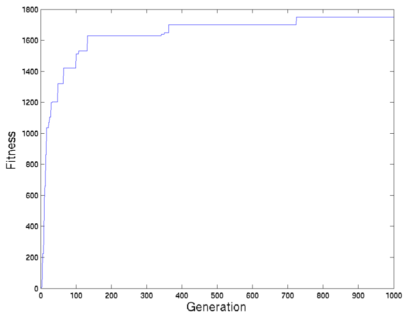
Extra experiment results

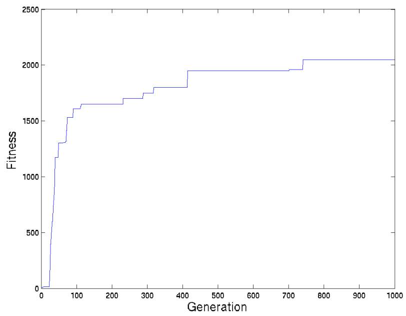

Fitness
Generation

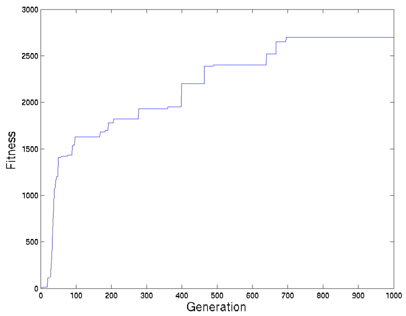

Fitness
Generation

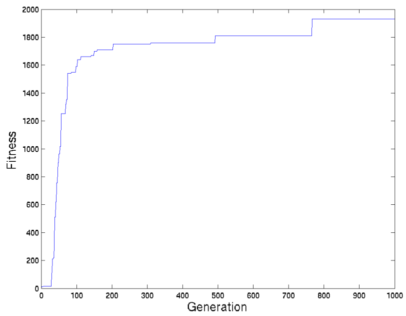

Fitness
Generation

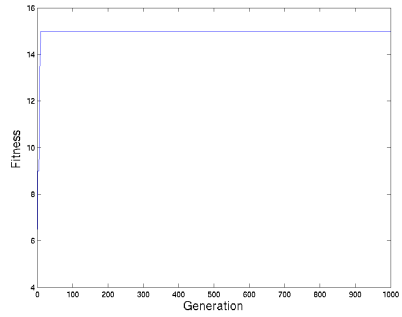

Fitness
Generation

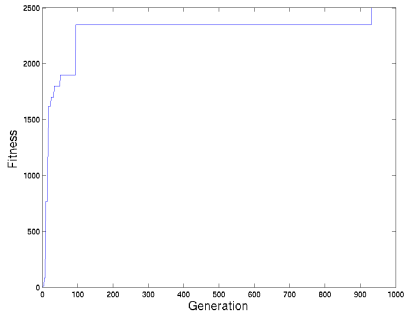

Fitness
Generation


Fitness
Generation


Fitness
Generation

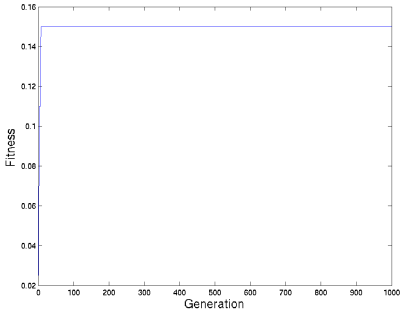

Fitness
Generation

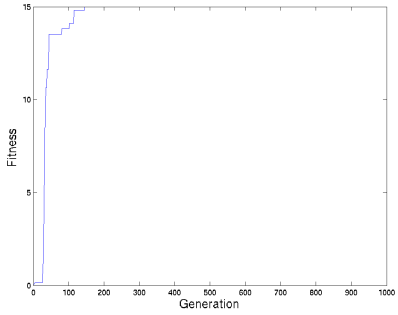


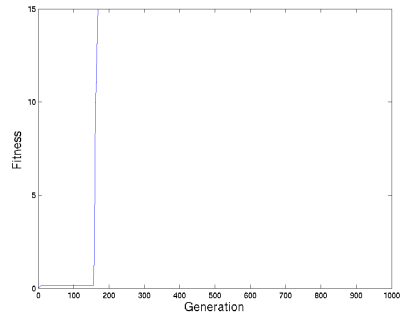


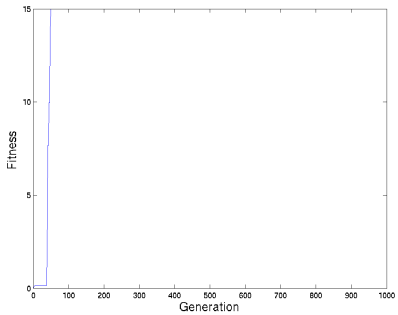
D.1.0.4 Strategic controller

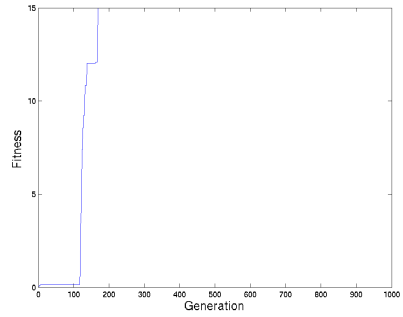
Main evolutionary process

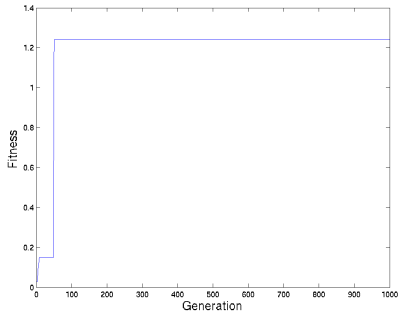


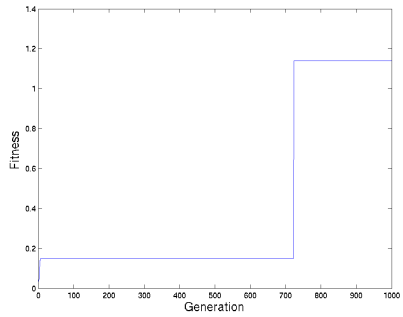


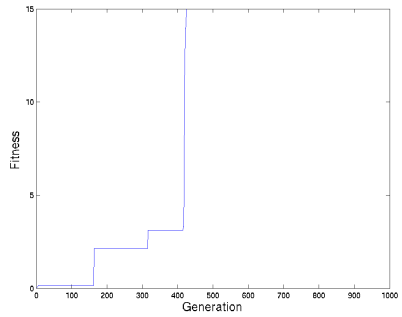


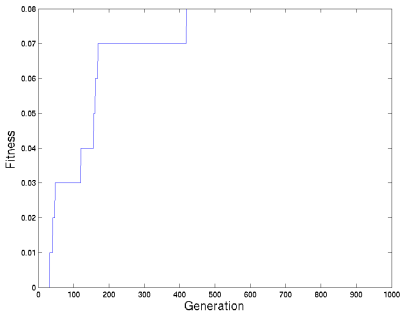


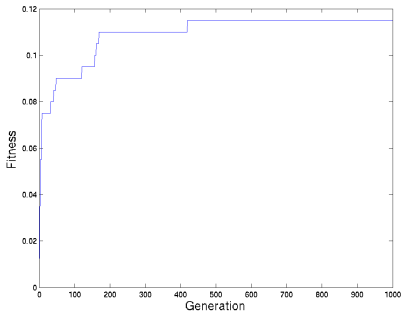




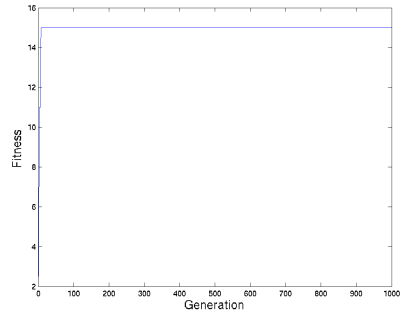


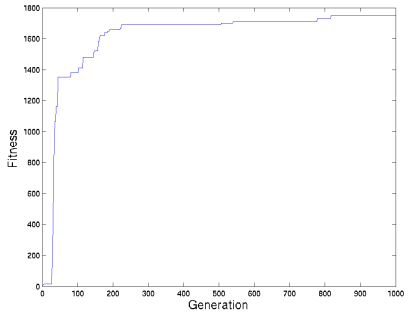


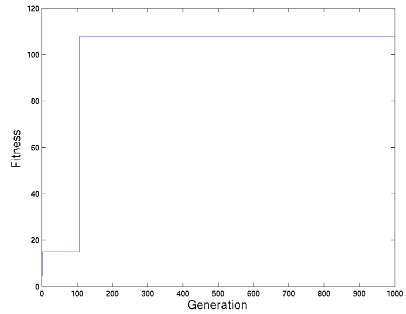


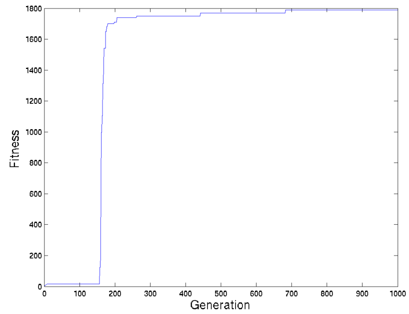


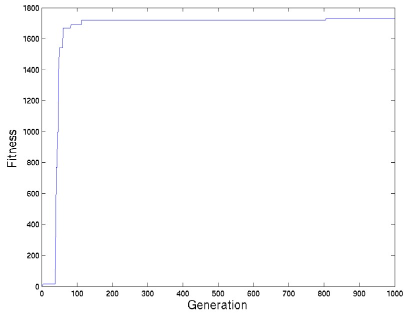
Extra experiment results

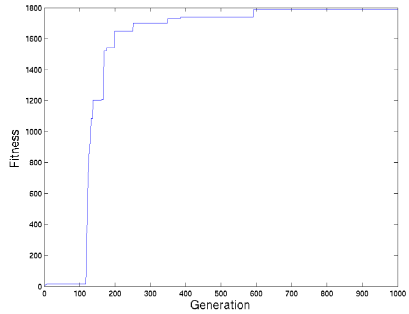


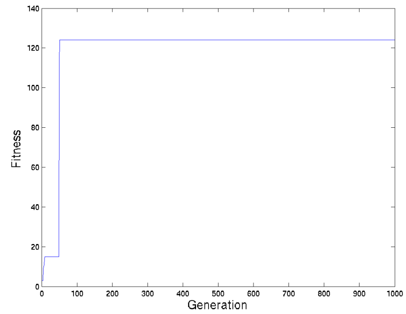


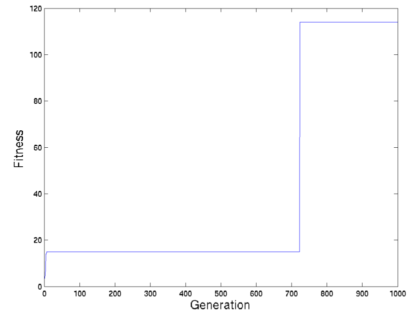

Fitness
Generation

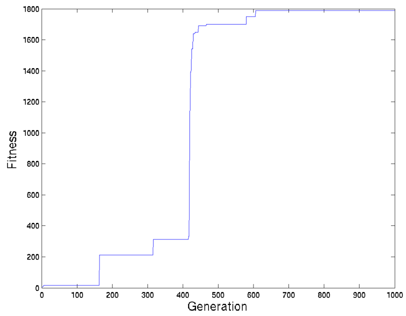

Fitness
Generation

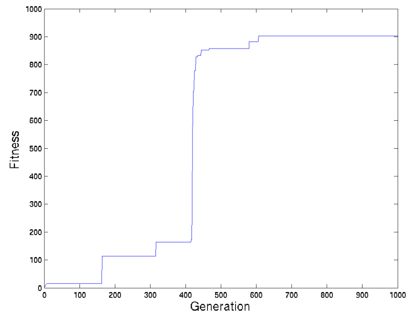

Fitness
Generation


Fitness
Generation


Fitness
Generation

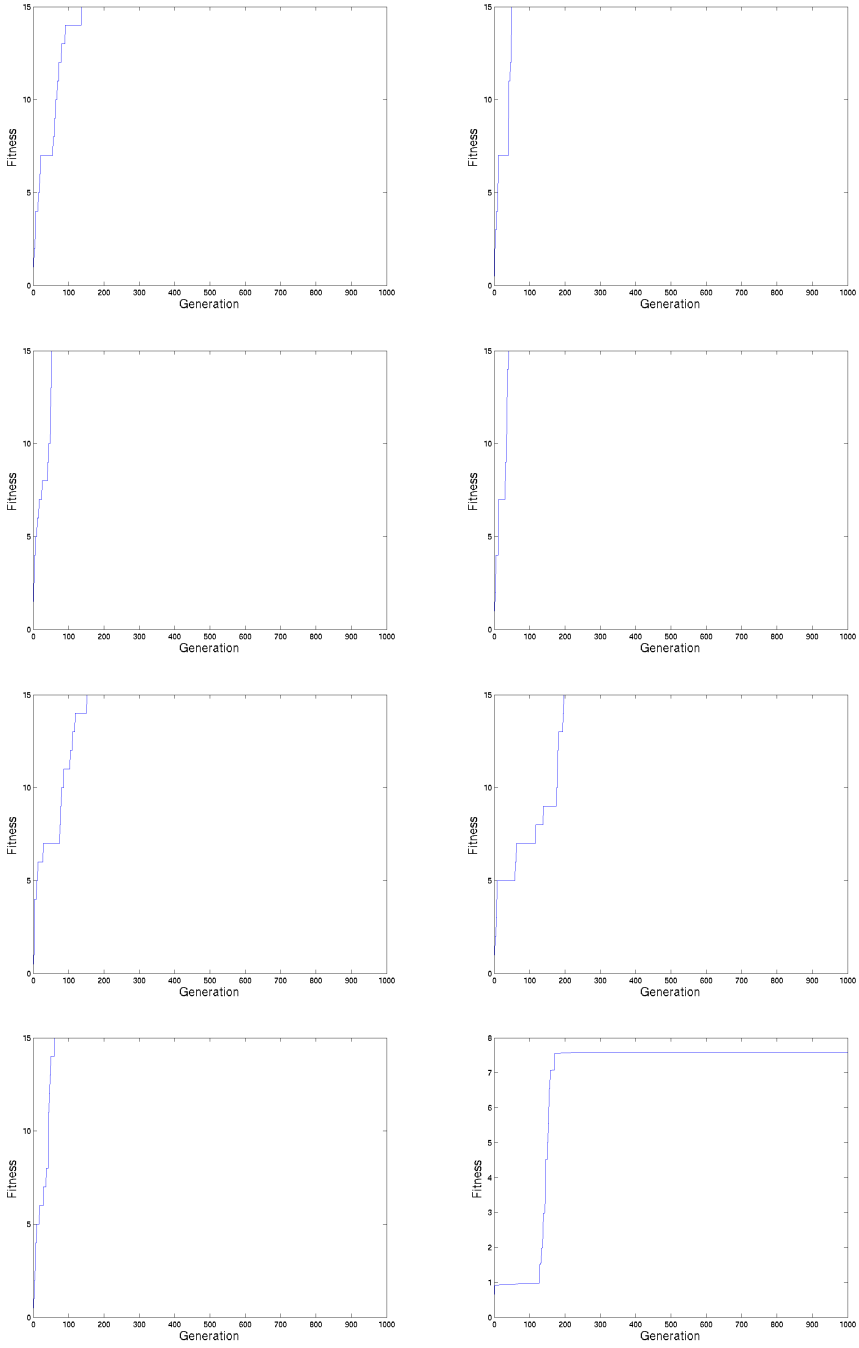

Fitness
Generation

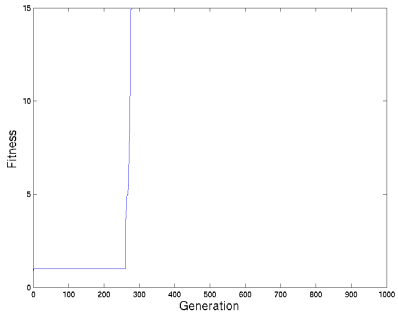

Fitness
Generation

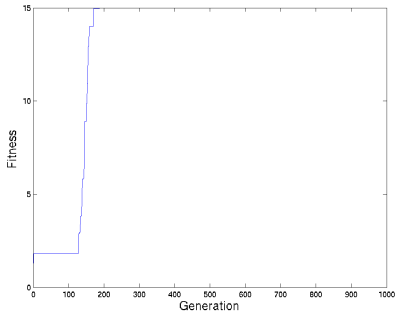

Fitness
Generation

D.1.0.5 Strategic and tactical controller

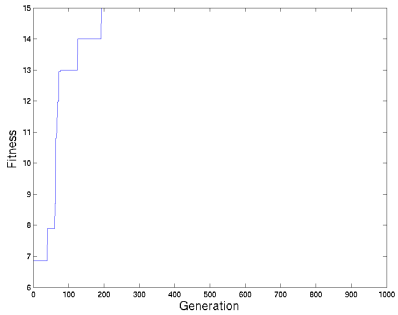
Picking the stick

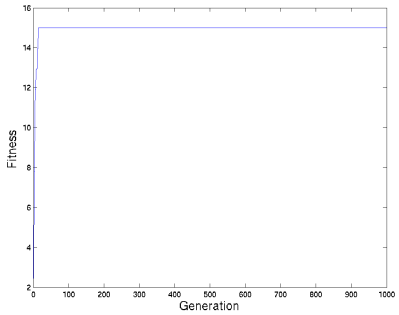


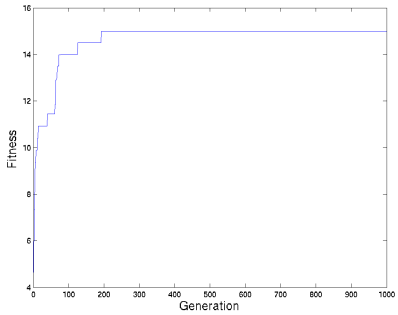


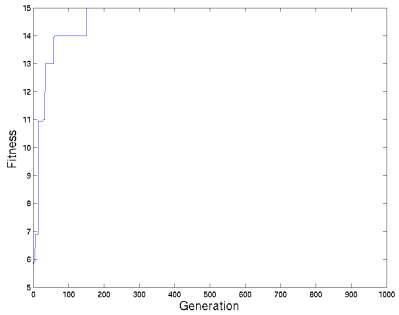


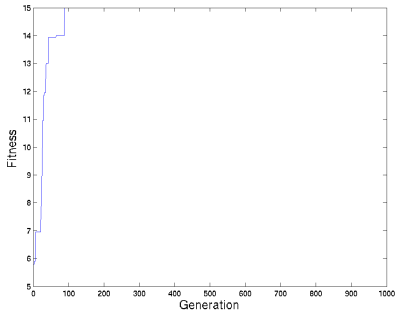
Releasing the stick

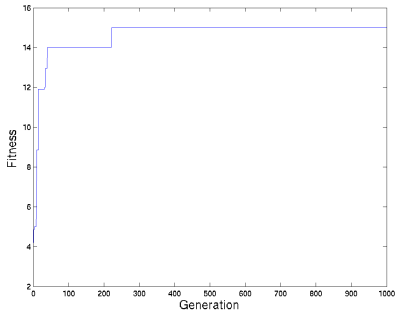


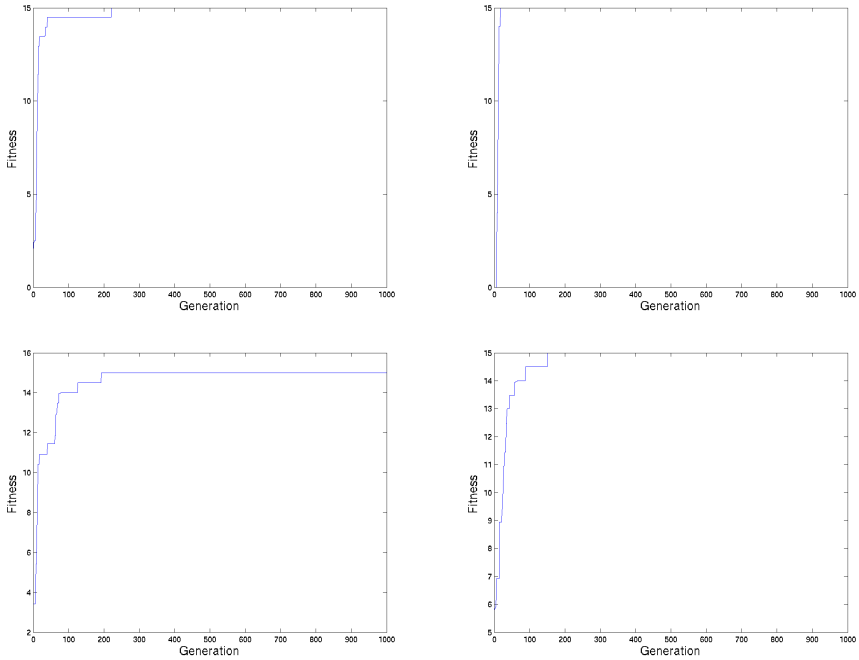






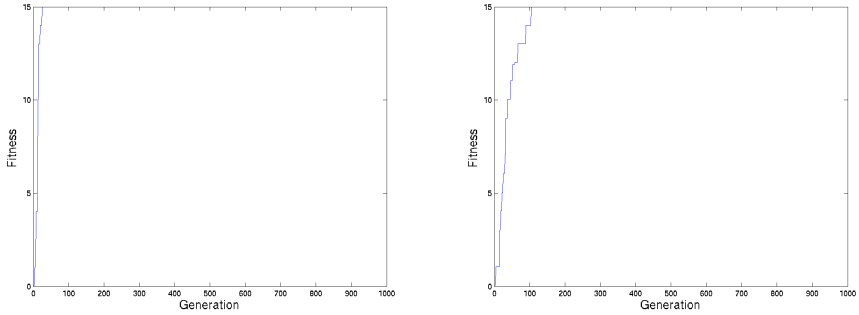


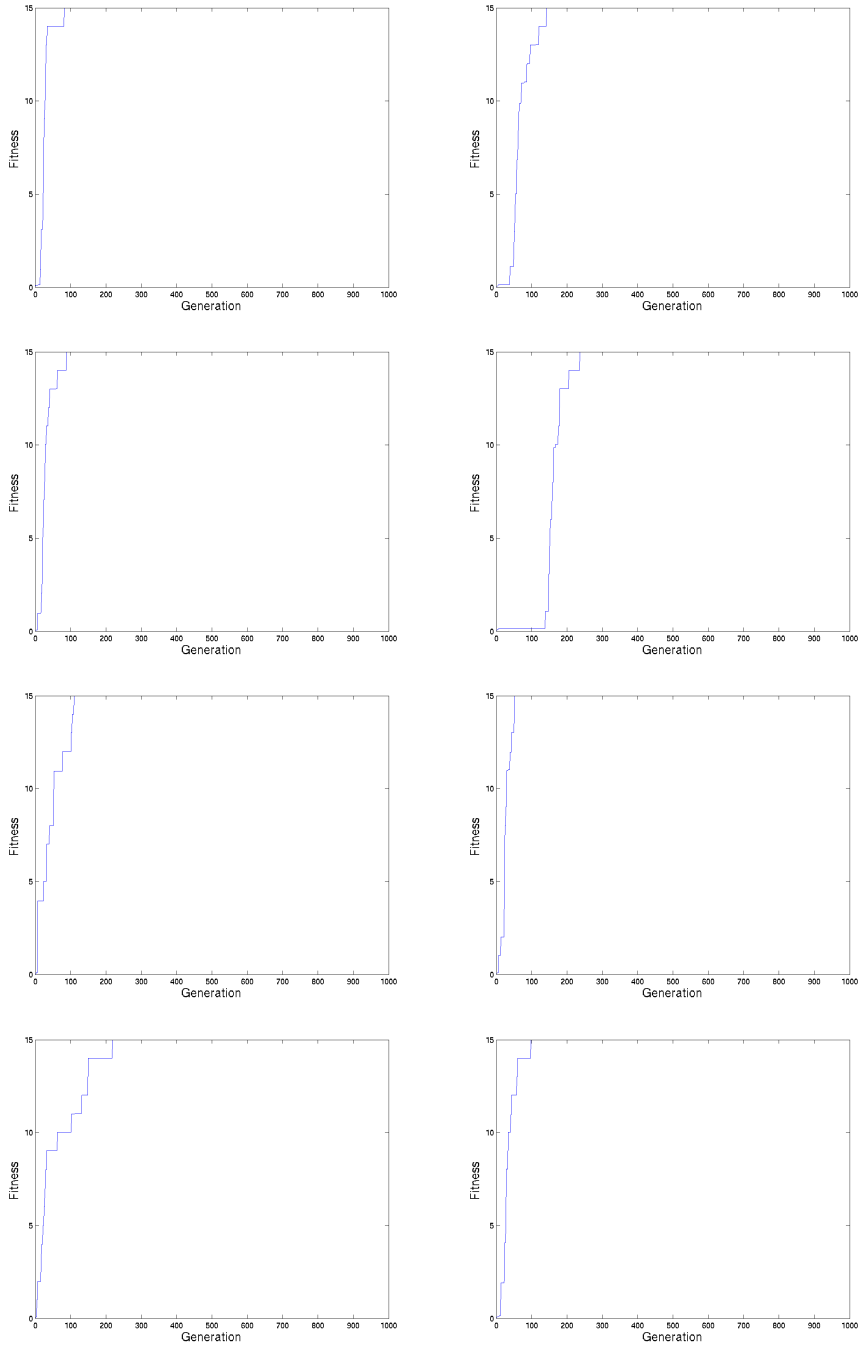




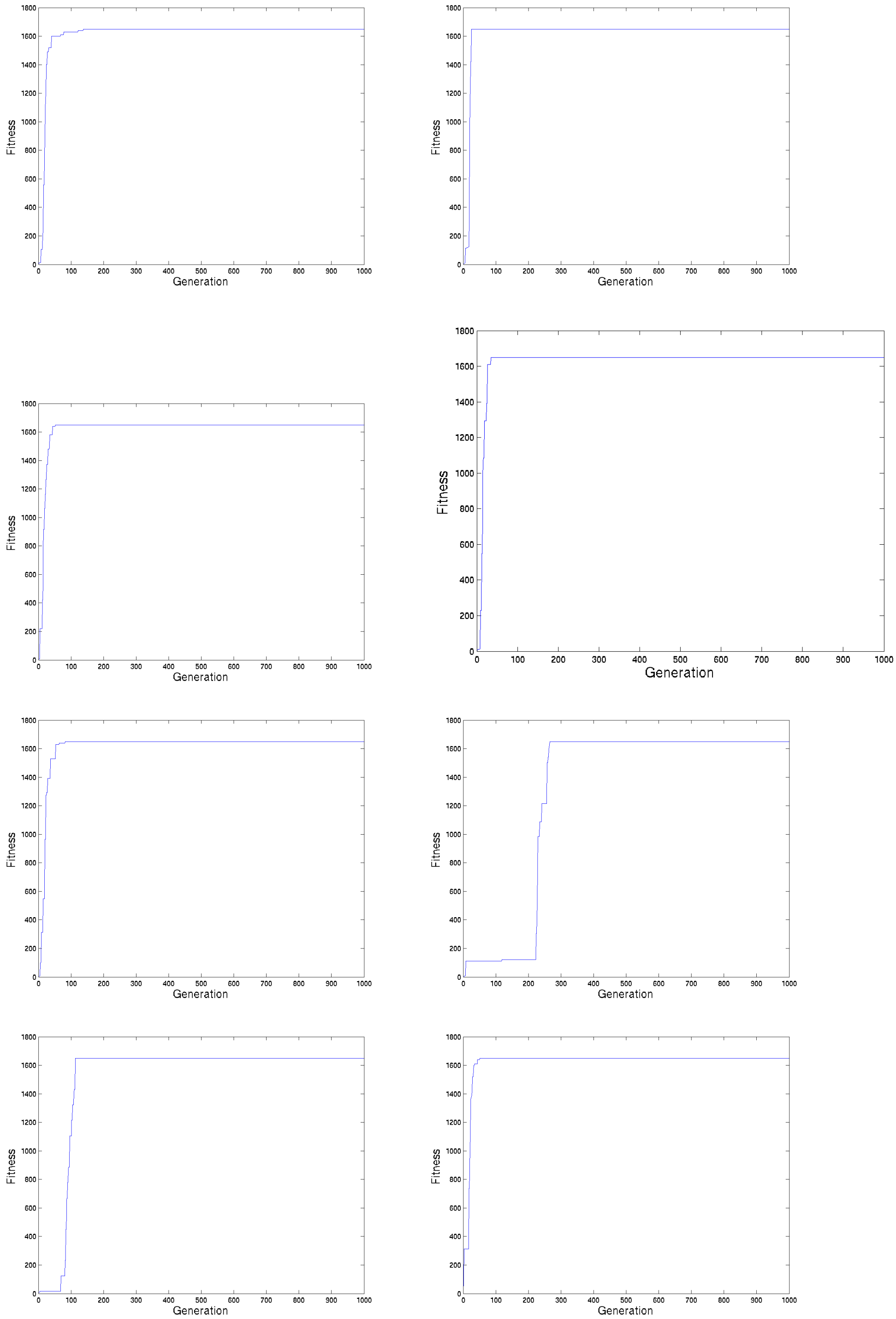
D.1.0.6 Emergent modular controller

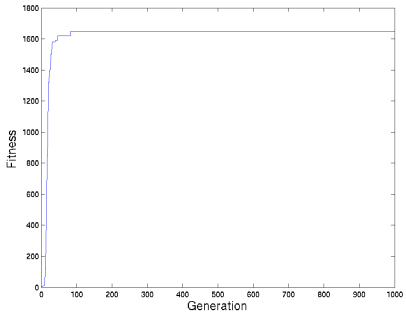
Main evolutionary process

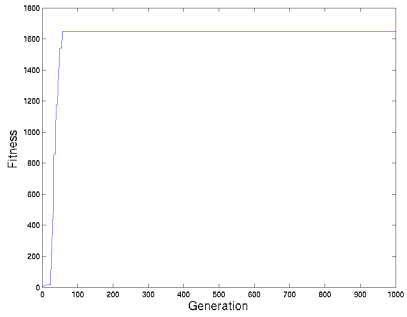




Extra experiment results

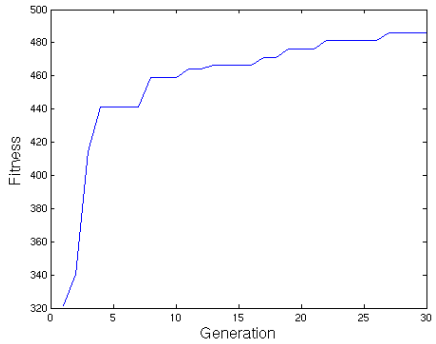

Fitness
Generation
0 200 400 600 800 1000 1200 1400 1600 1800
0 100 200 300 400 500 600 700 800 900 1000

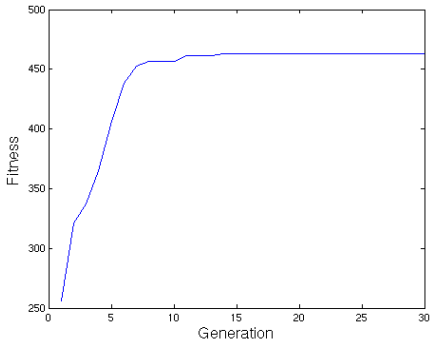


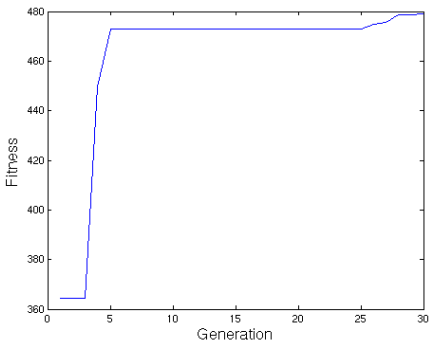


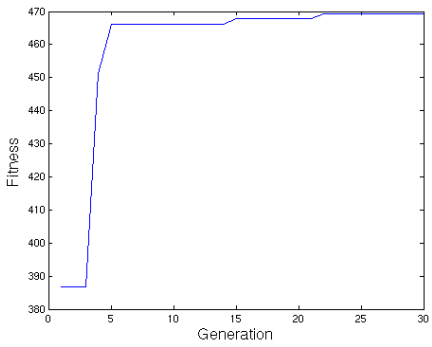
D.2 Experiment results for the different architectures in the Aibo stand up test

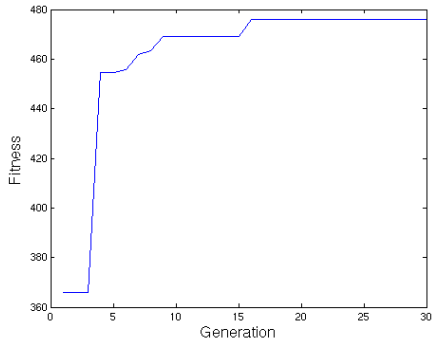
D.2.0.7 Monolithic controller

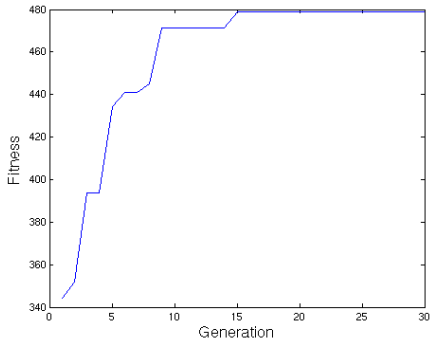


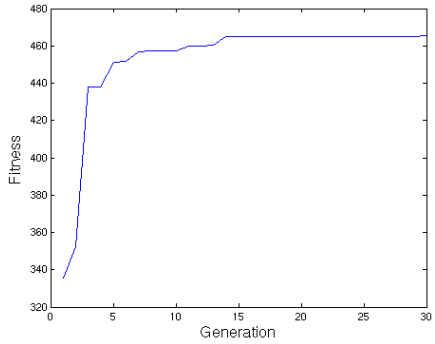


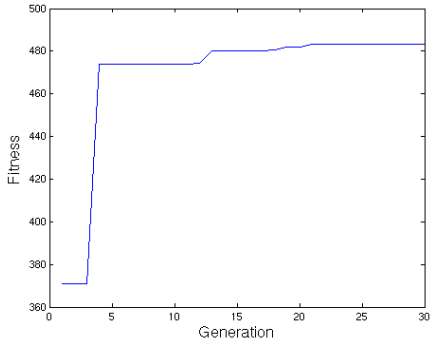


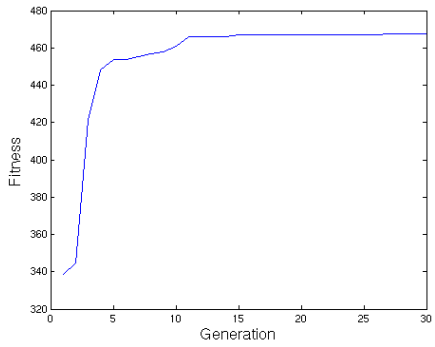


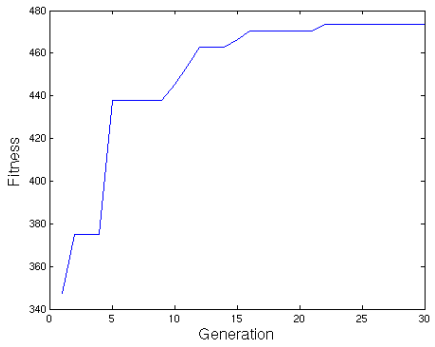

Fitness
Generation


Fitness
Generation

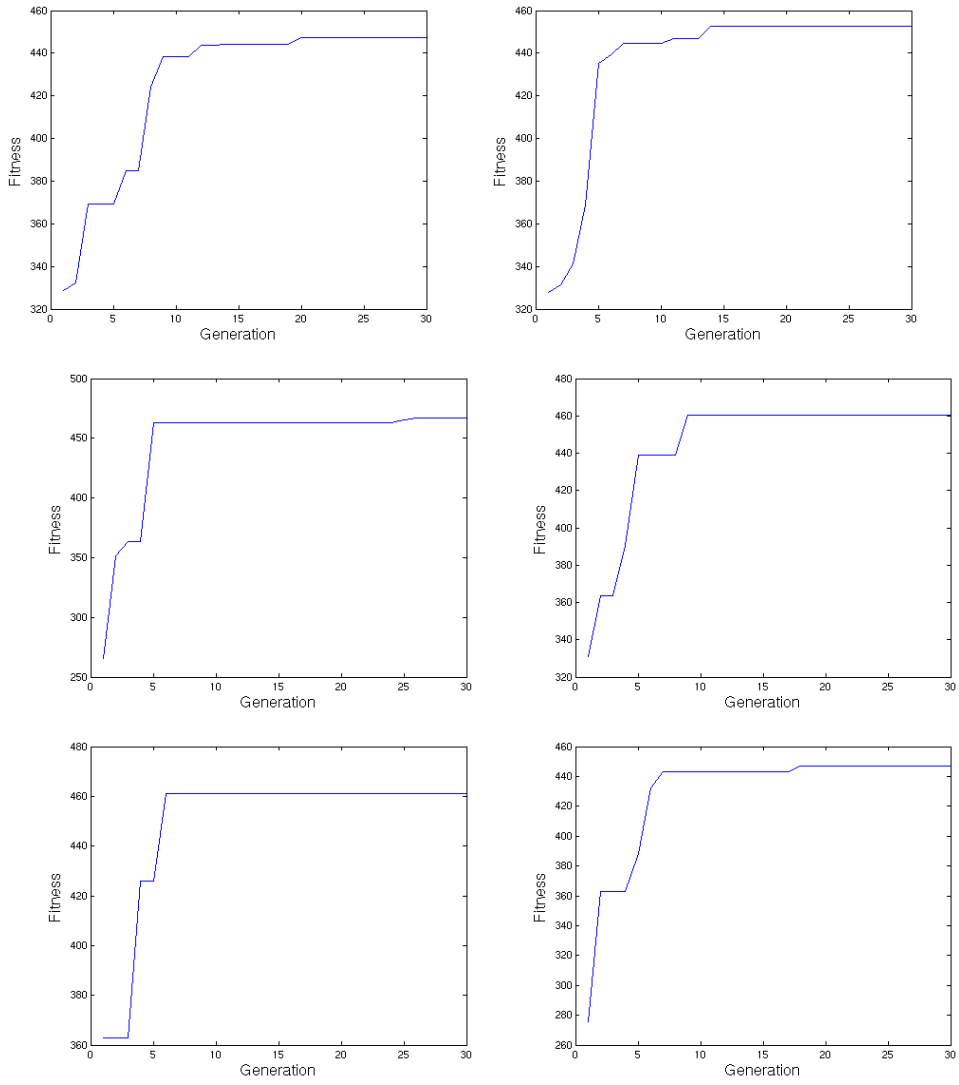

Fitness
Generation

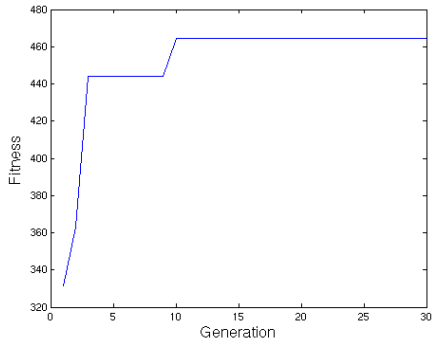

Fitness
Generation

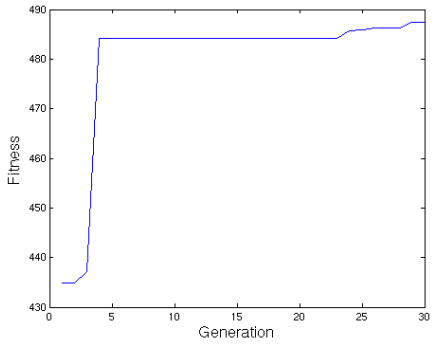

Fitness
Generation

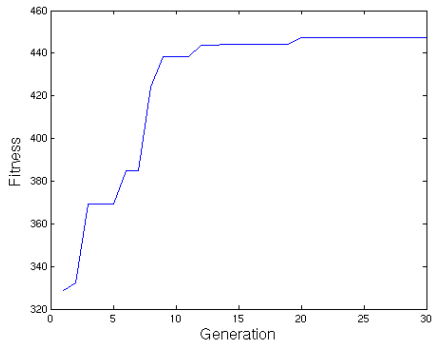

Fitness
Generation

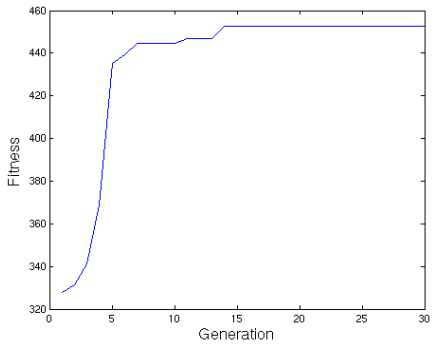
D.2.0.8 Tactical controller

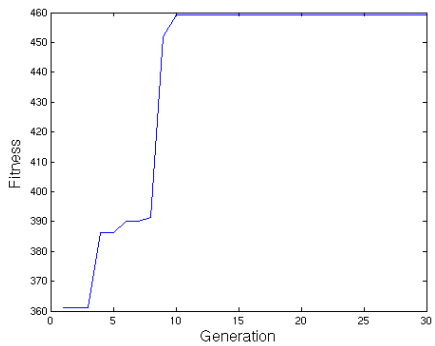


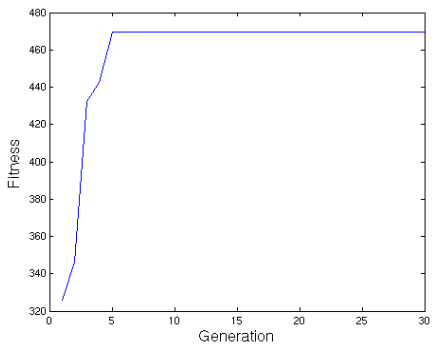

Fitness
Generation


Fitness
Generation

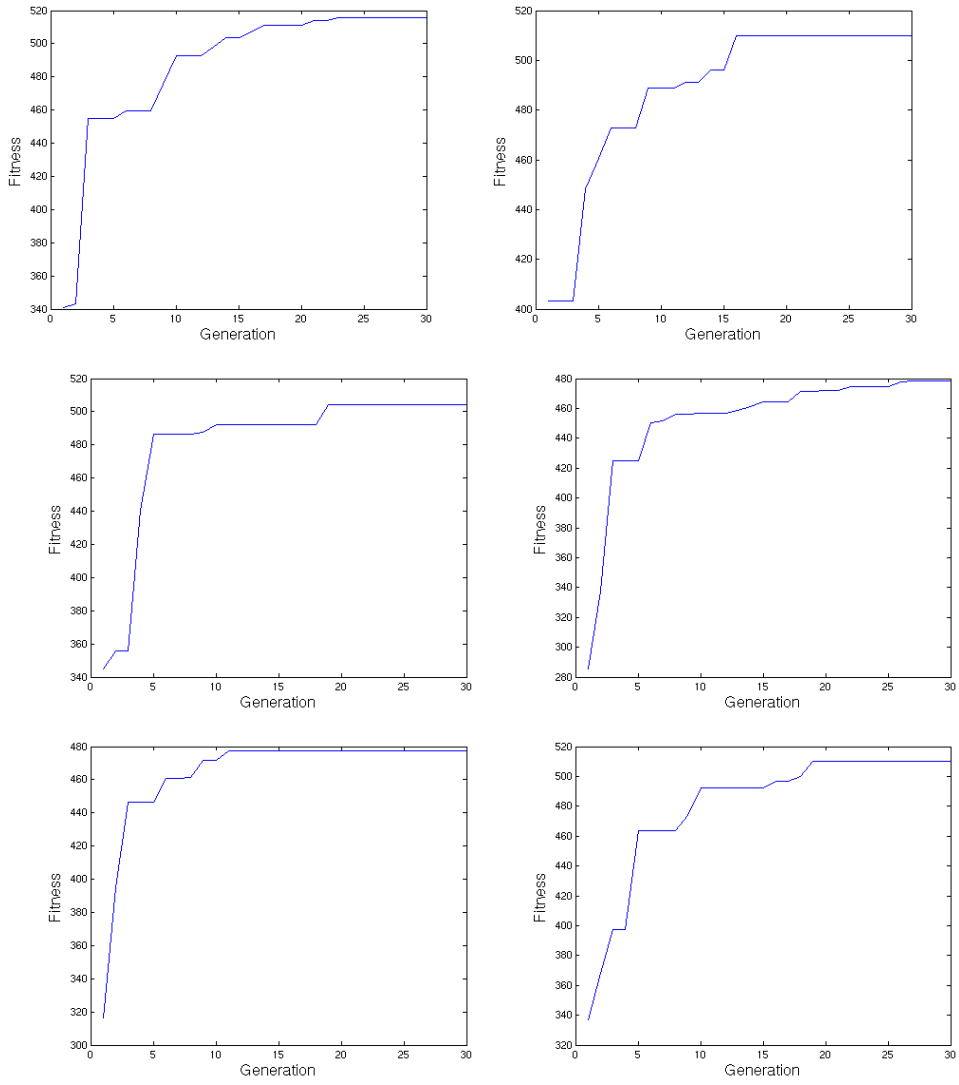

Fitness
Generation

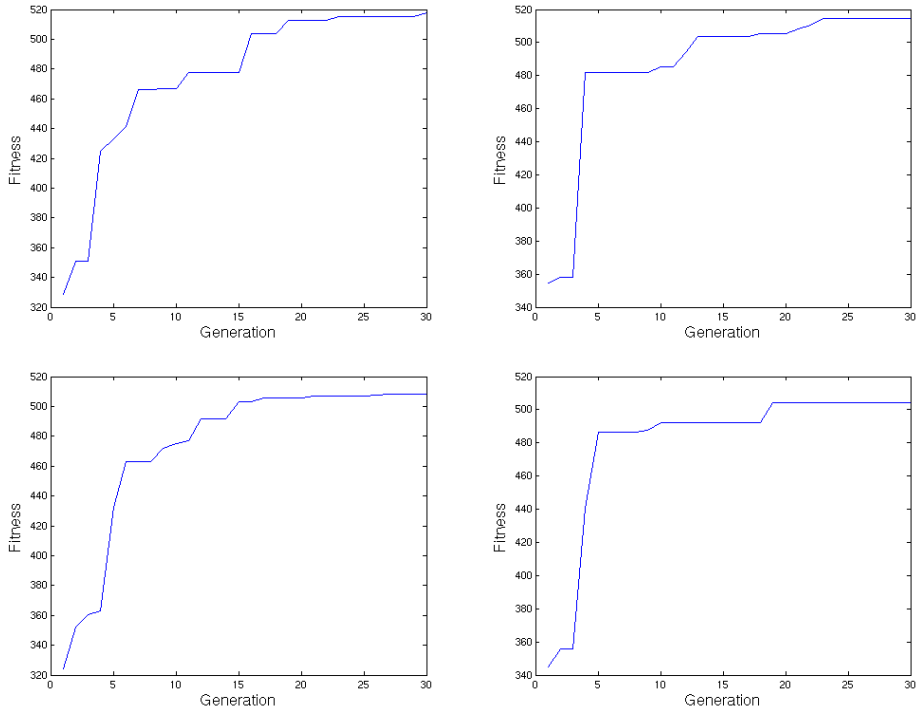

Fitness
Generation


Fitness
Generation


Fitness
Generation

D.2.0.9 Emergent modular controller




Fitness
Generation
Fitness
Generation
Fitness
Generation
Fitness
Generation

E

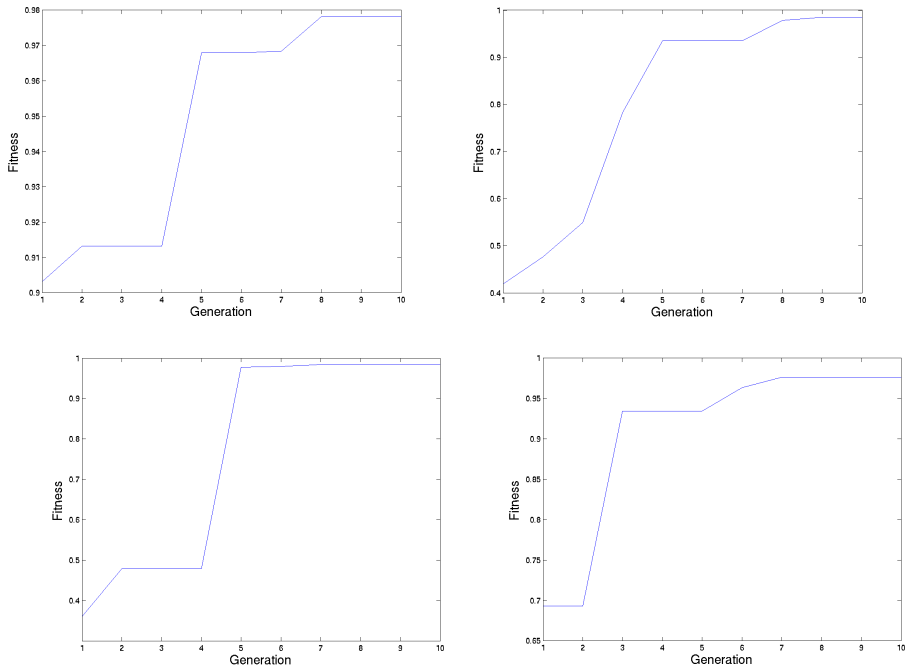
Chapter 6 results

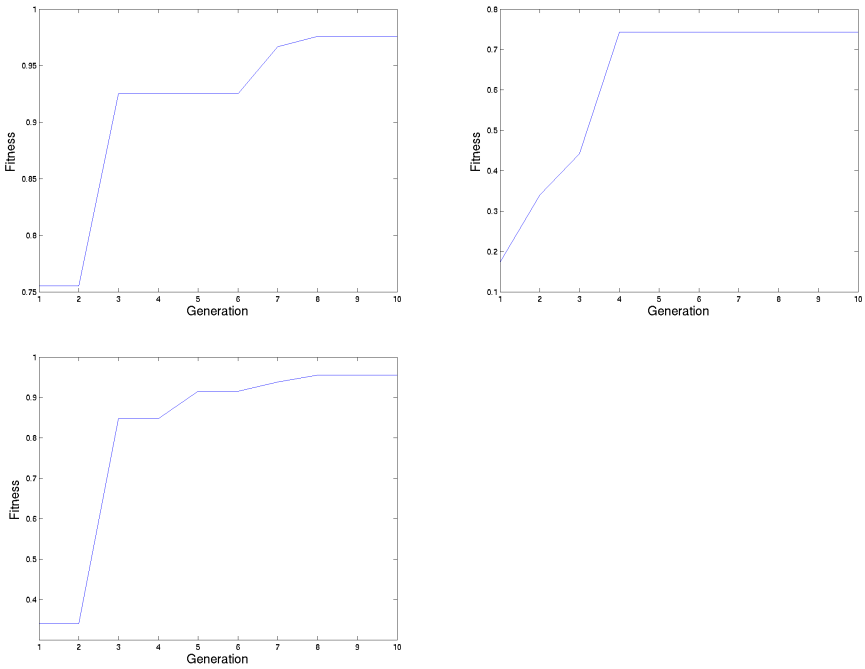
E.1 Experiment results for the progressive design experiments

This sections contains the evolution of the fitness for all the ten runs performed for the evolution of each type of controller of chapter 6.

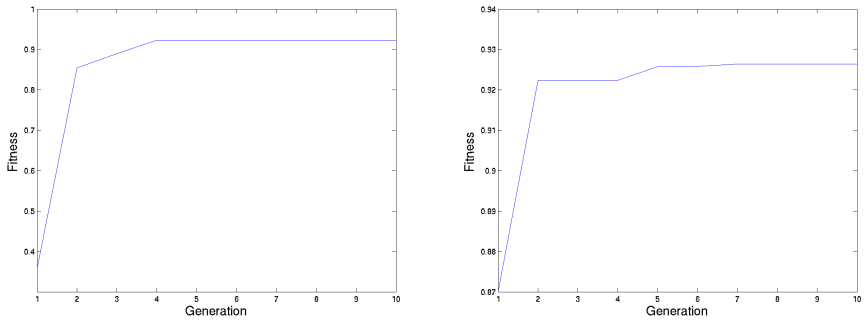
E.1.1 The Aibo walking controller

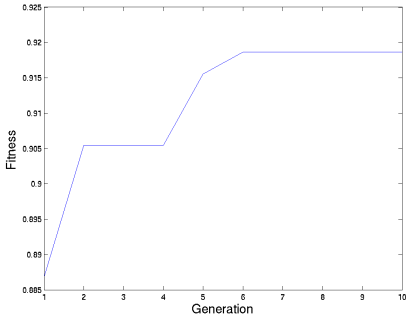
E.1.1.1 Progressive design first stage (J1 joint)

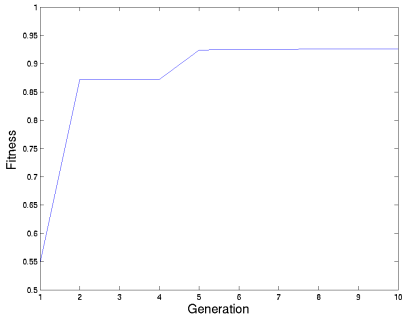


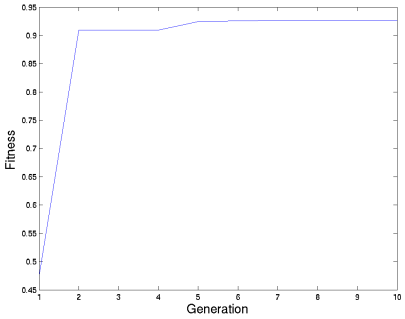


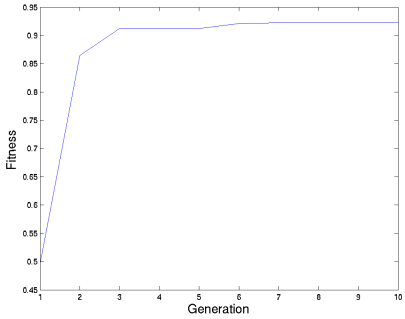
E.1.1.2 Progressive design first stage (J2 joint)

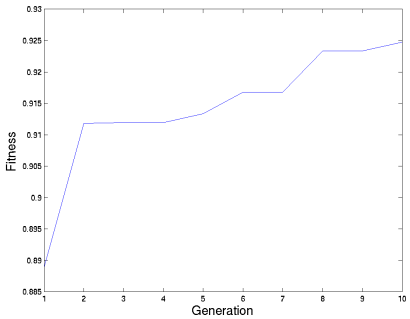



Fitness
Generation

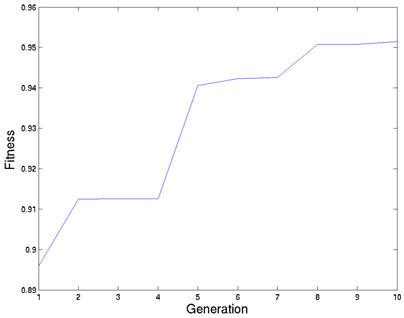

Fitness
Generation

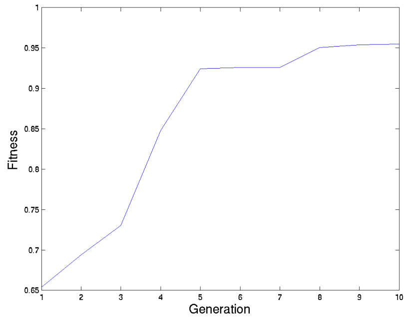

Fitness
Generation

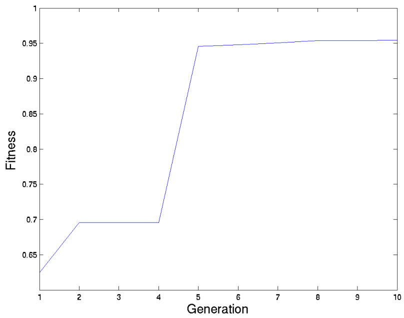

Fitness
Generation

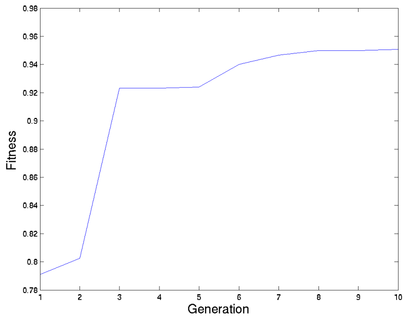

Fitness
Generation

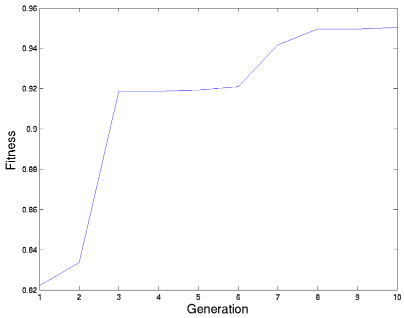
E.1.1.3 Progressive design first stage (J3 joint)

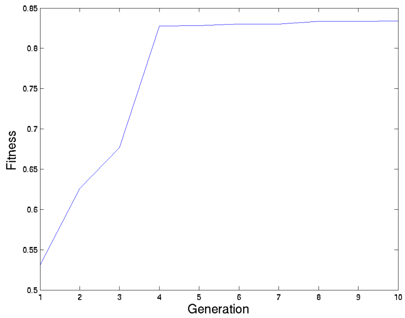


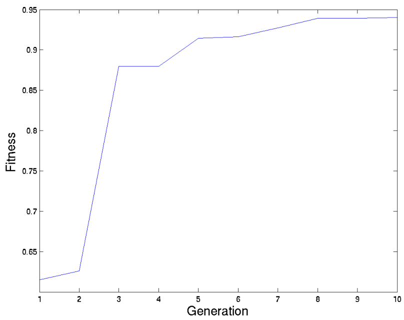




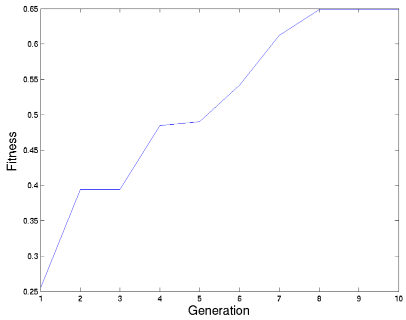


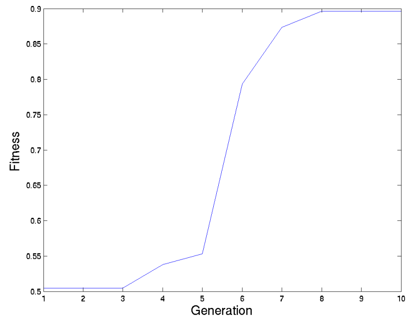


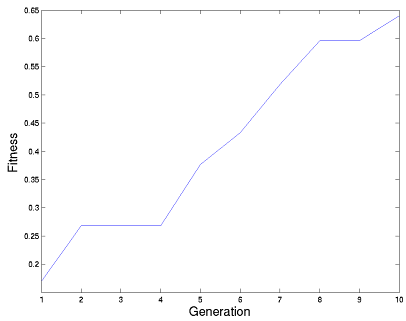


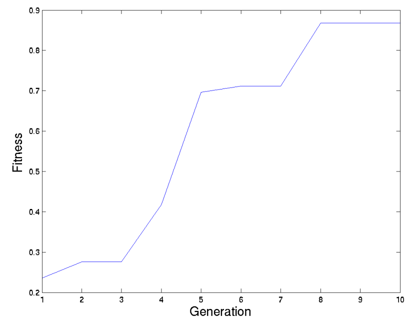


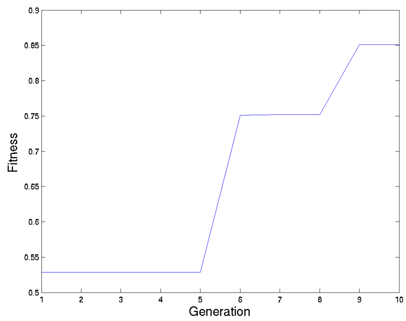
E.1.1.4 Progressive design second stage (J1 joints)

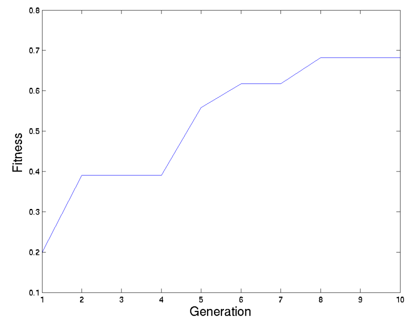


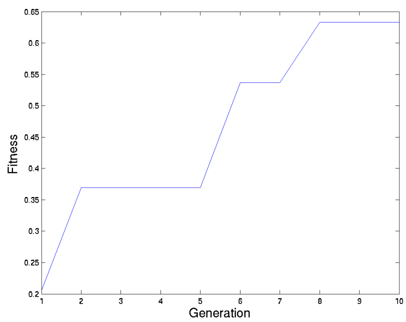




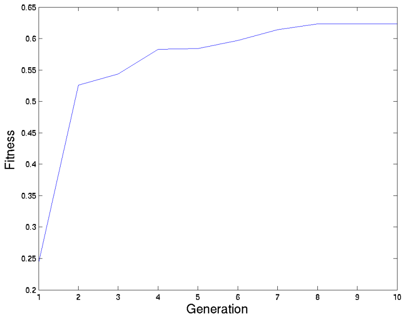


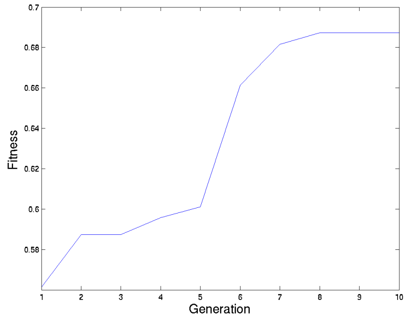


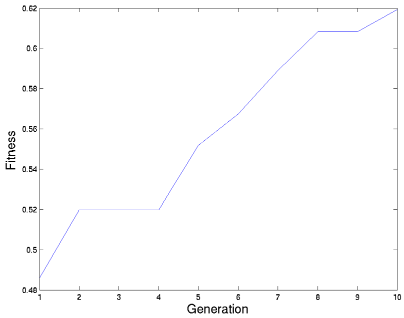


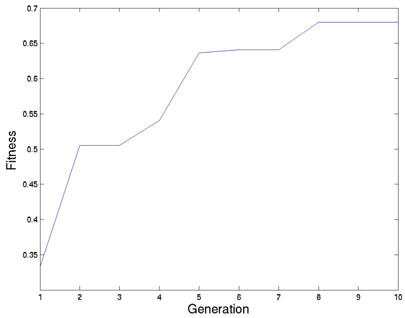


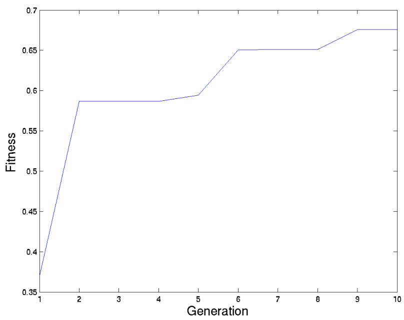
E.1.1.5 Progressive design second stage (J2 joints)

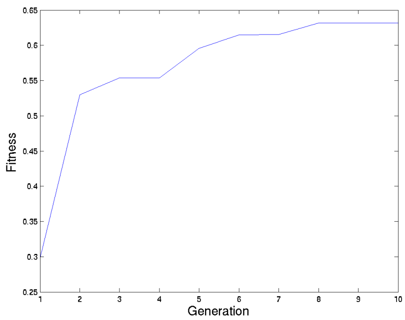


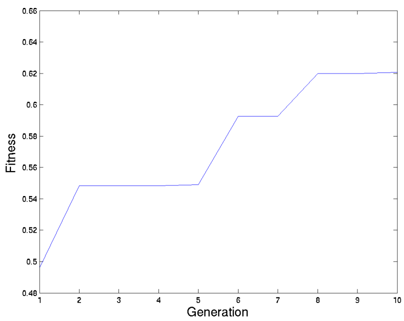




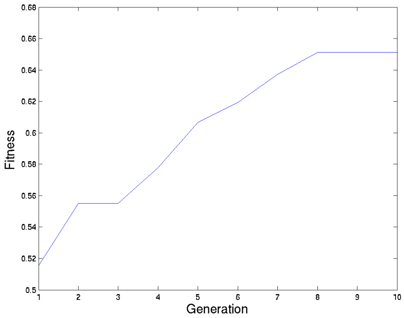


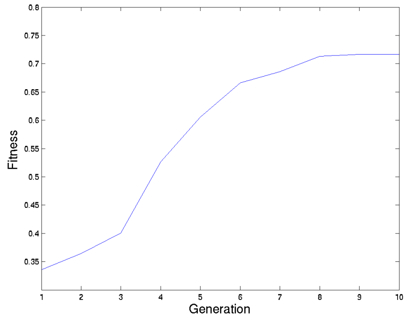


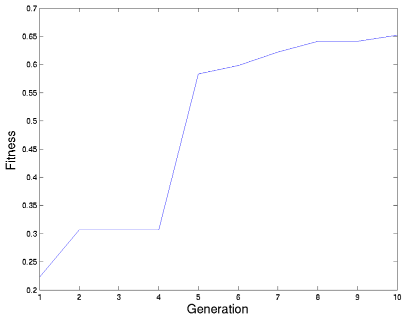


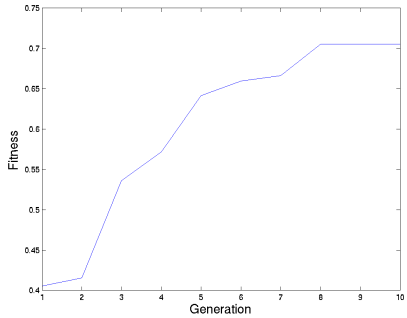


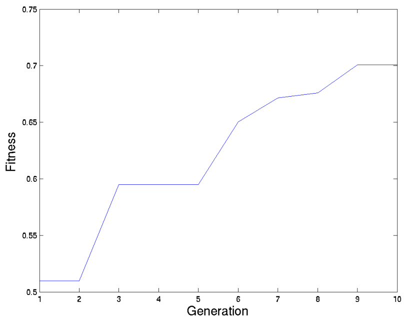
E.1.1.6 Progressive design second stage (J3 joints)

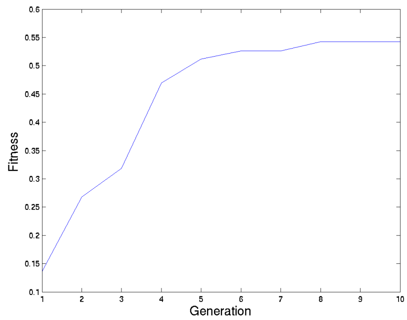


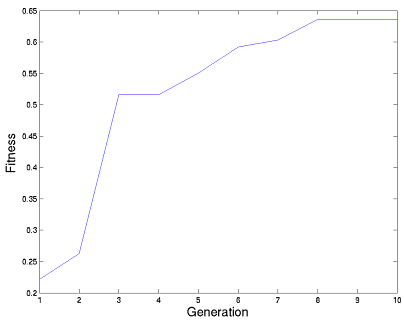




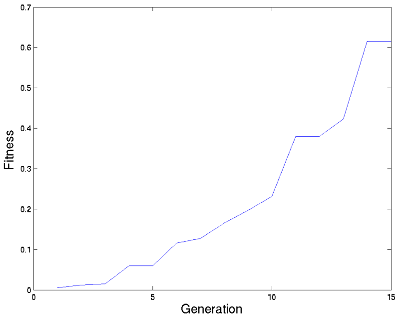


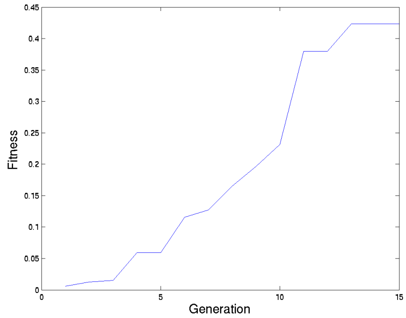


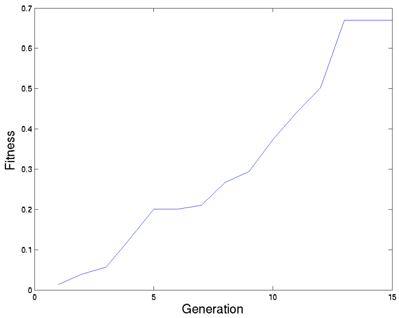


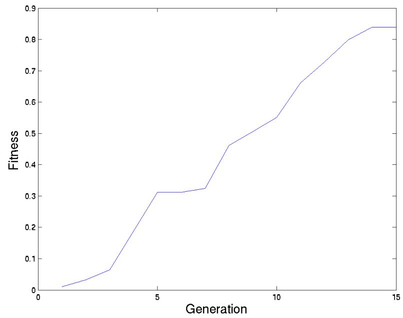


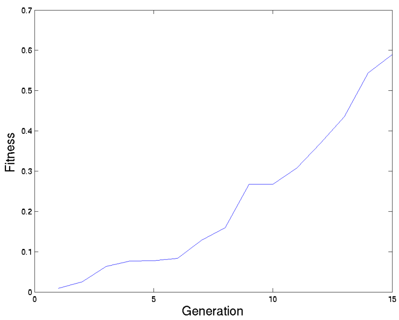
E.1.1.7 Progressive design third stage (J1 joints)

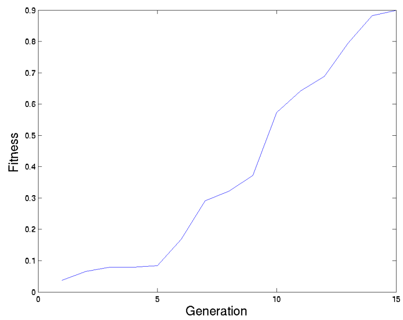


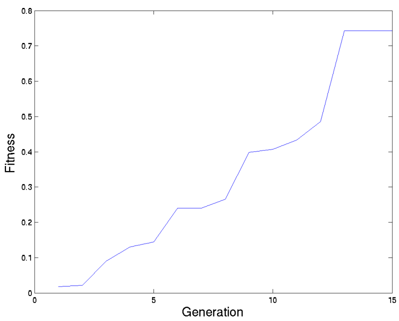




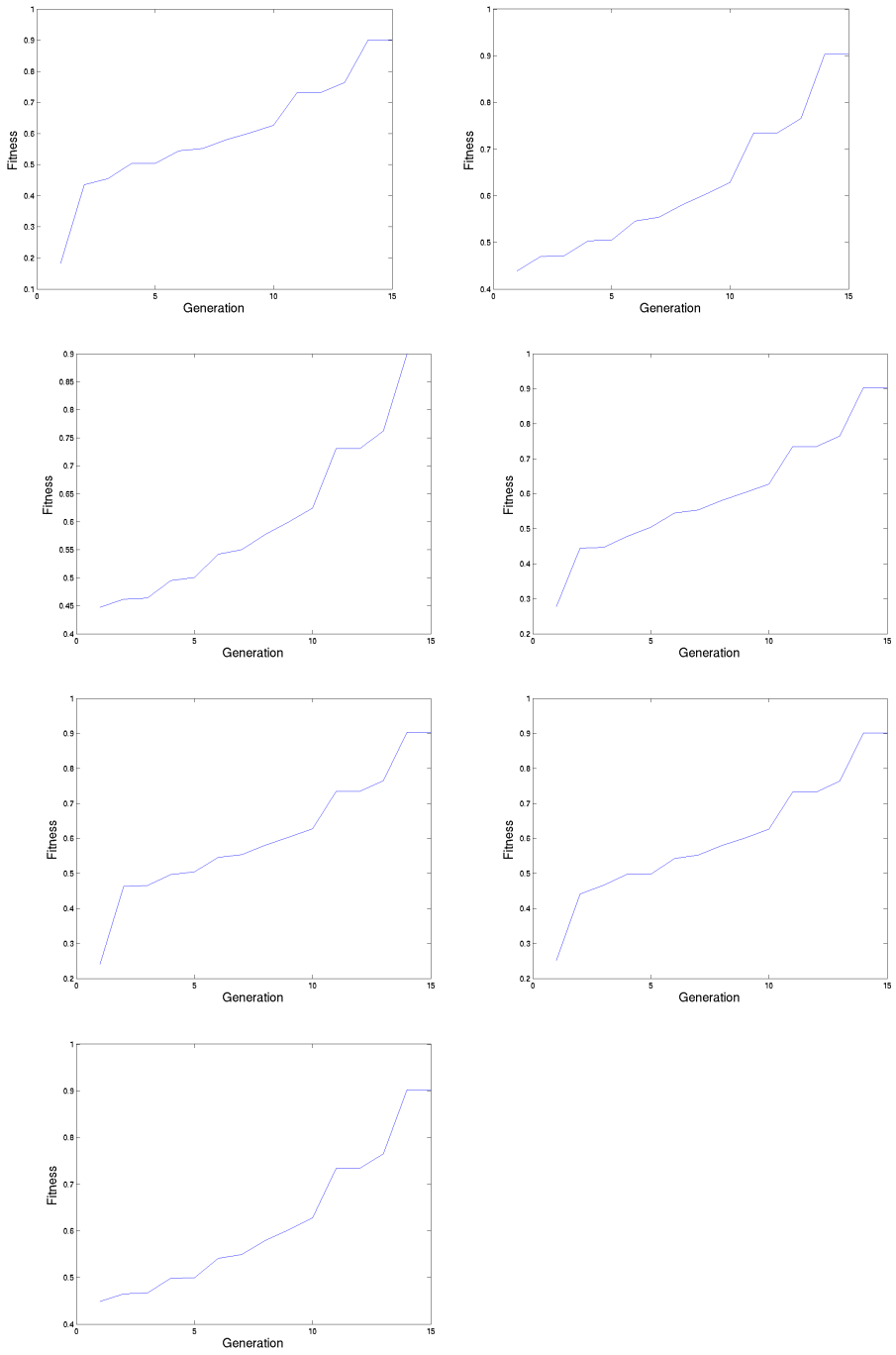




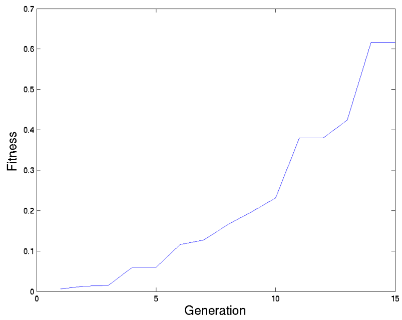


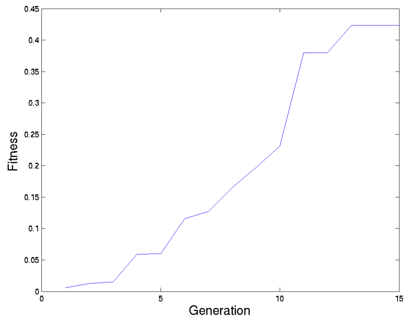


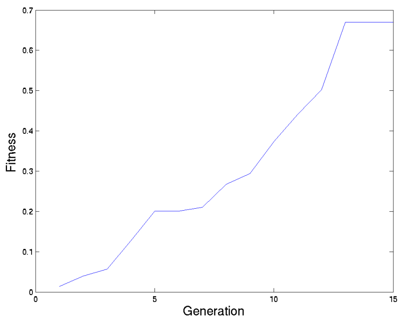
E.1.1.8 Progressive design third stage (J2 joints)

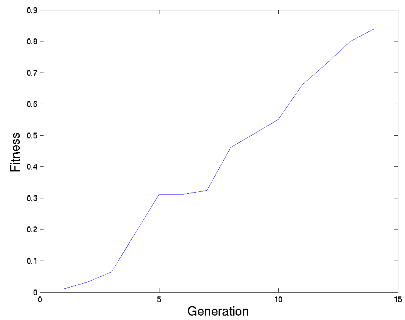


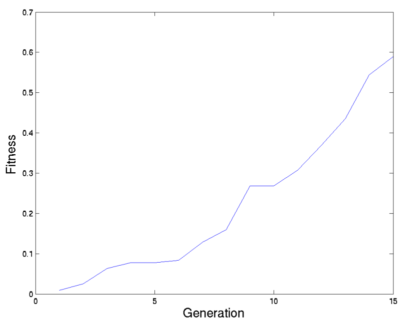
E.1.1.9 Progressive design third stage (J3 joints)

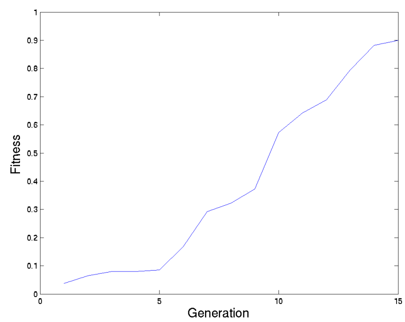


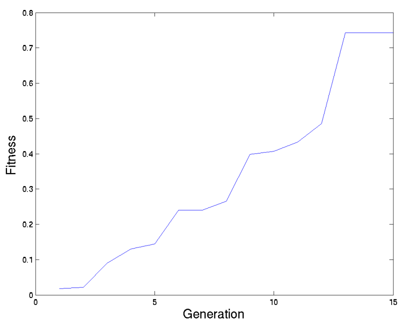












F

Chapter 8 results

F.1 Frequency analysis of the garbage collector DAIR controller internal states

This sections contains the plots of the internal state signals generated by the DAIR garbage collector controller, for its analysis of chapter 8. It includes the frequency plots. Only the non-tonic states are plotted.

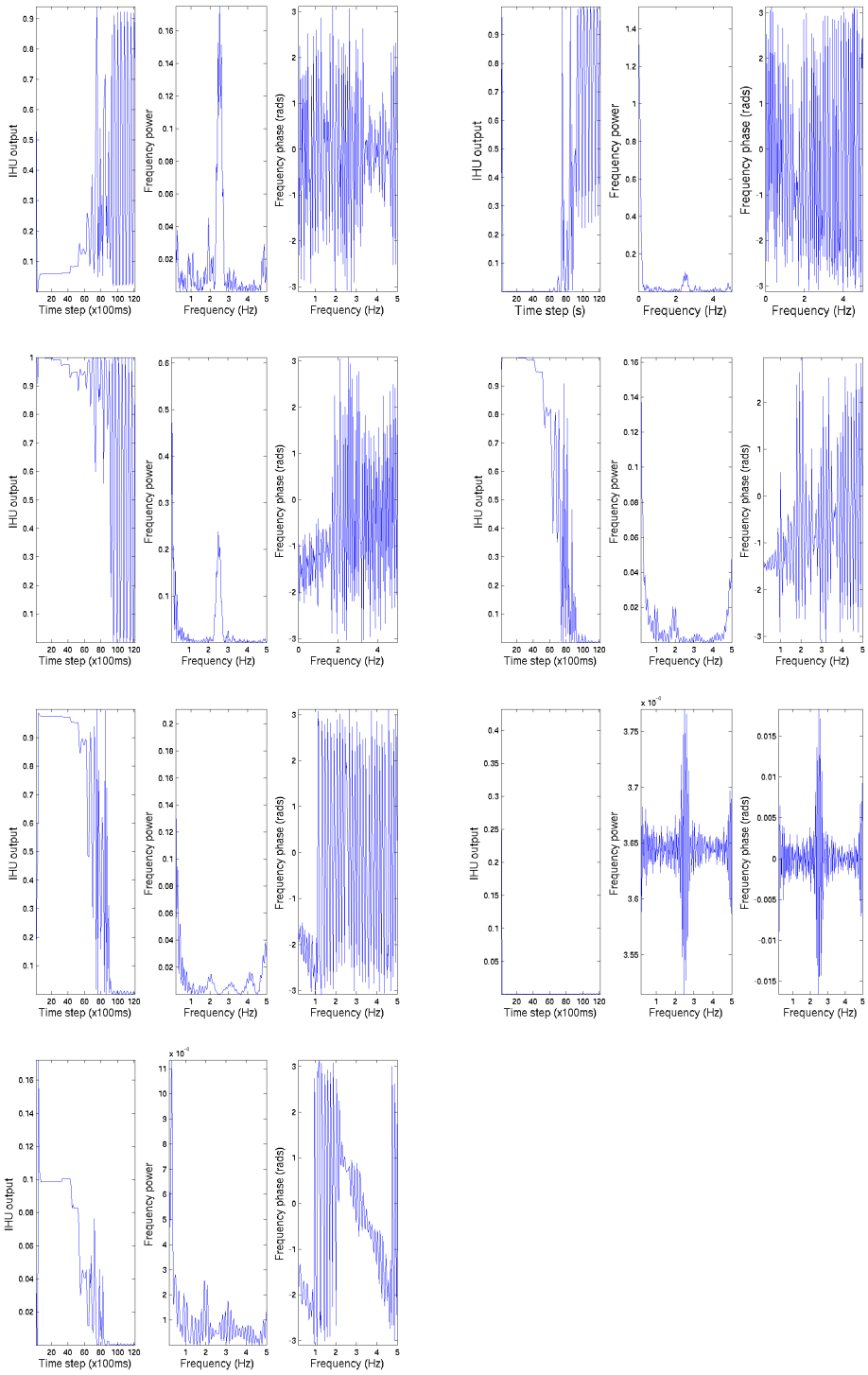


Figure F.1: Frequency plot of sensor IHUs in situation 5a. From left to right, top to down, sensor S_A, S_B, S_C, S_D, S_E, S_F, S_G.

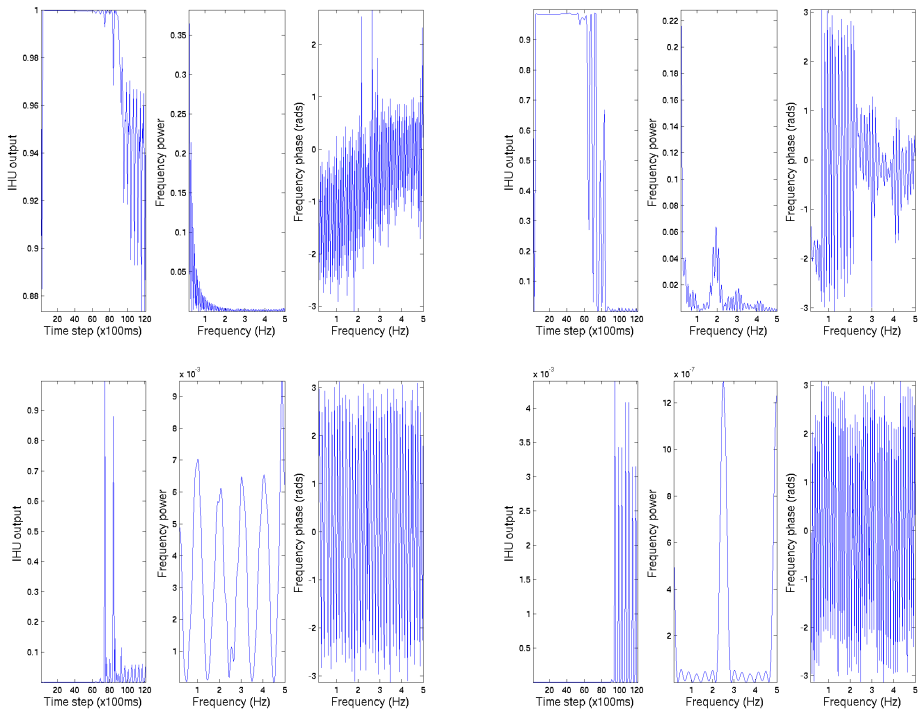


Figure F.2: Frequency plot of actuator IHUs in situation 5a. From left to right, top to down, sensor M_L, M_R, P_T, P_R.

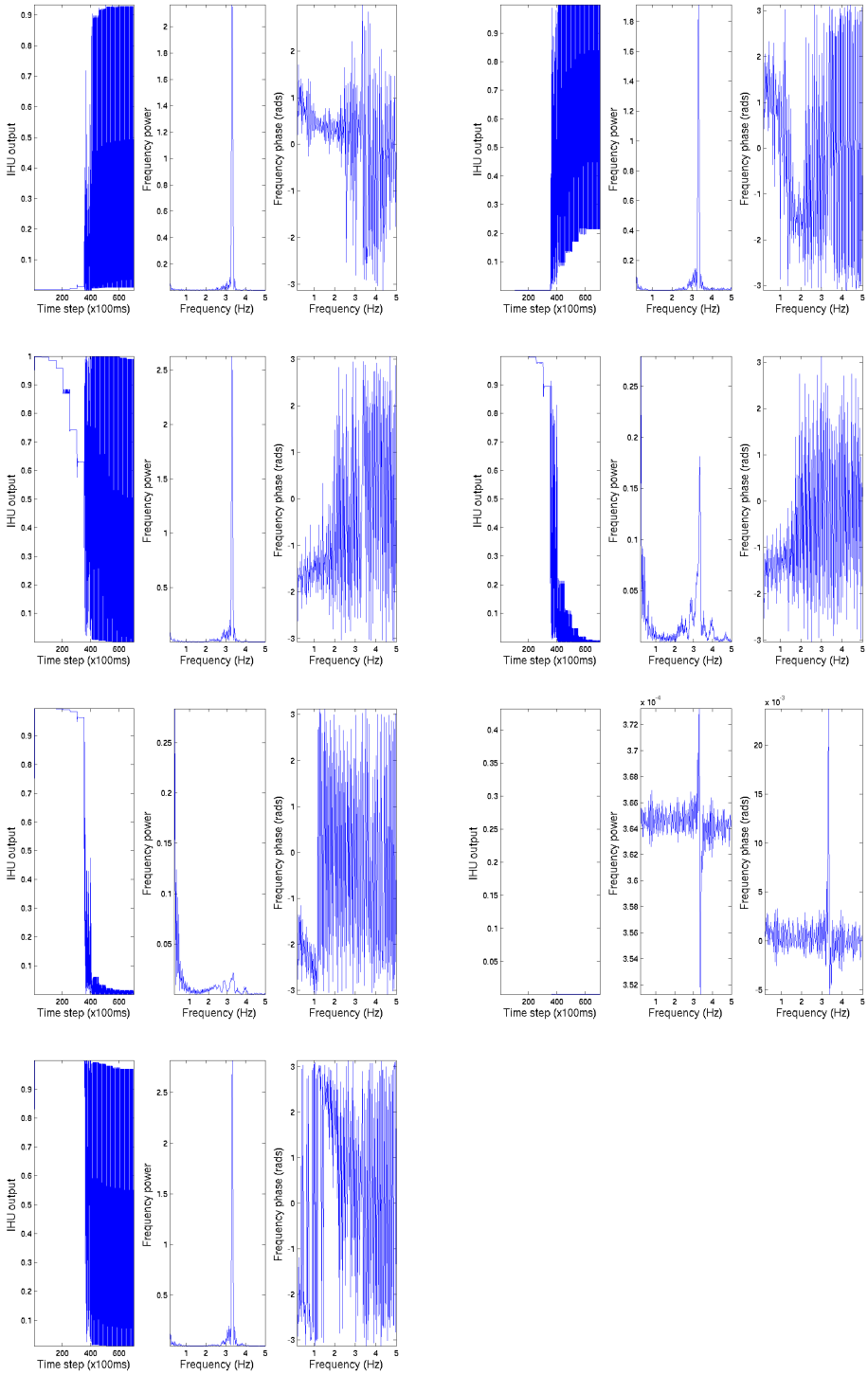


Figure F.3: Frequency plot of sensor IHUs in situation 5b. From left to right, top to down, sensor S_A, S_B, S_C, S_D, S_E, S_F, S_G.

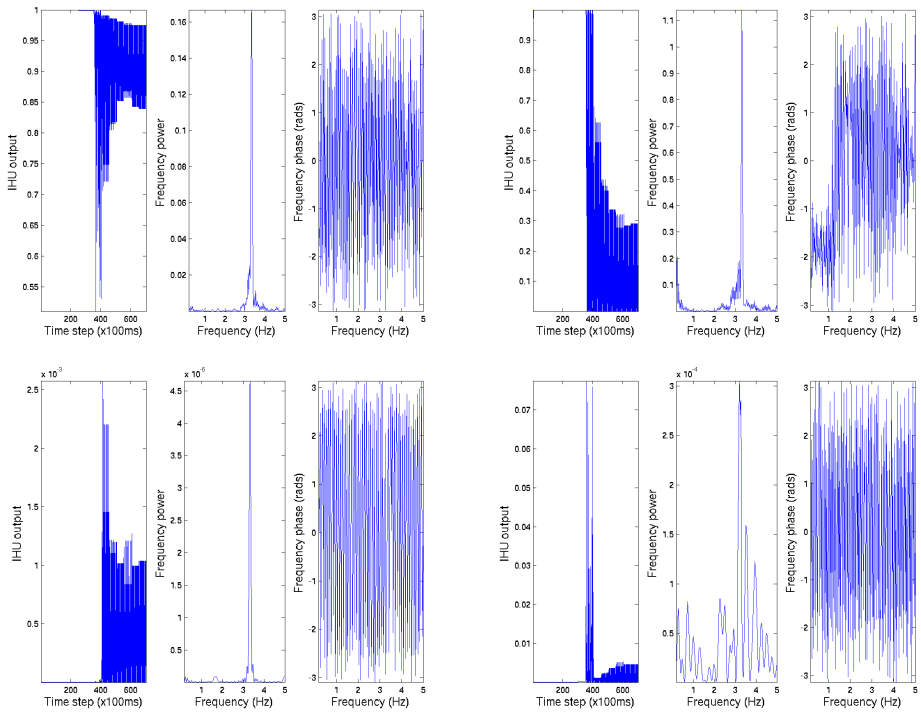


Figure F.4: Frequency plot of actuator IHUs in situation 5b. From left to right, top to down, sensor M_L, M_R, P_T, P_R.

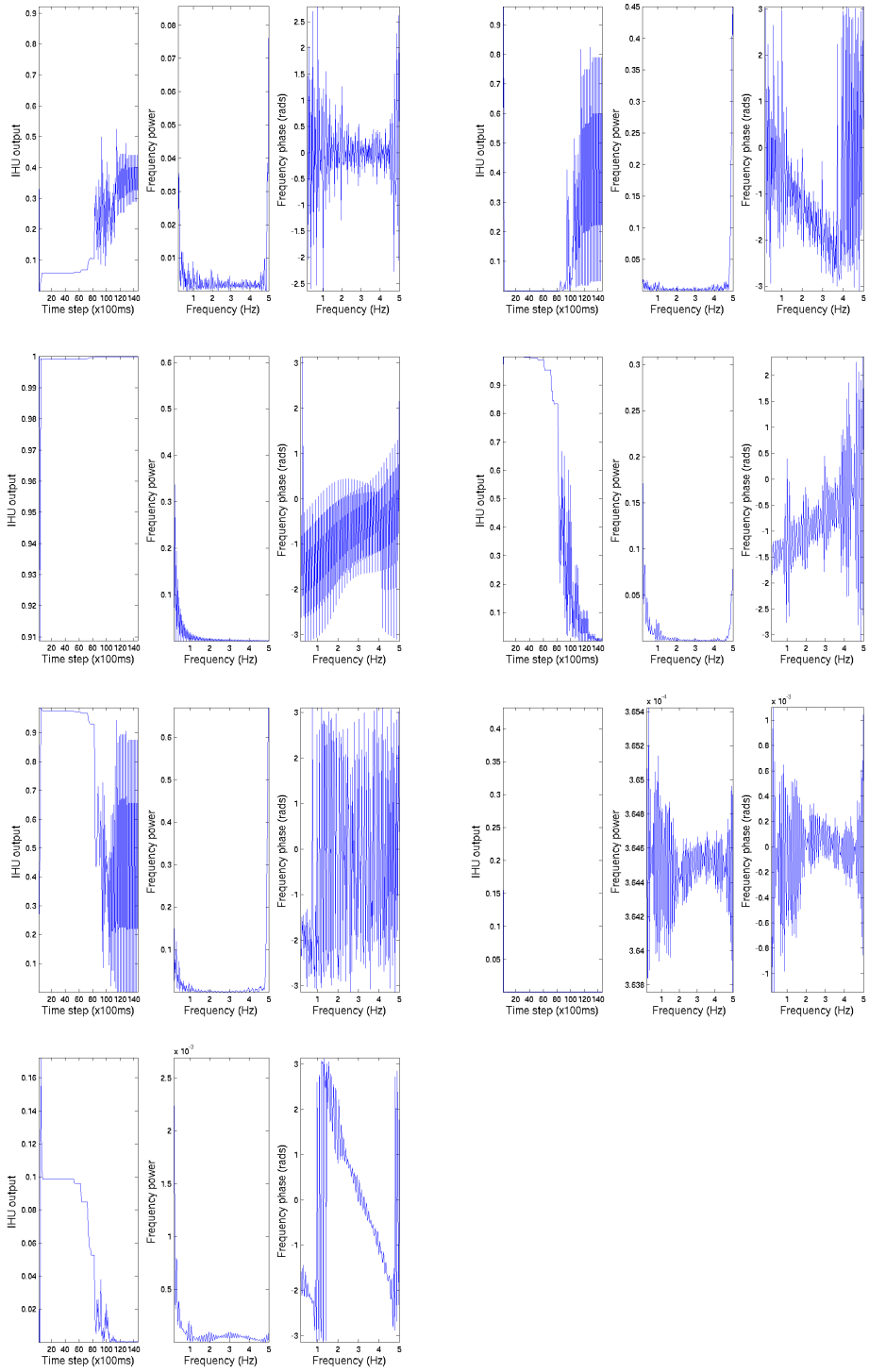


Figure F.3: Frequency plot of sensor IHUs in situation 6a. From left to right, top to down, sensor S_A, S_B, S_C, S_D, S_E, S_F, S_G.

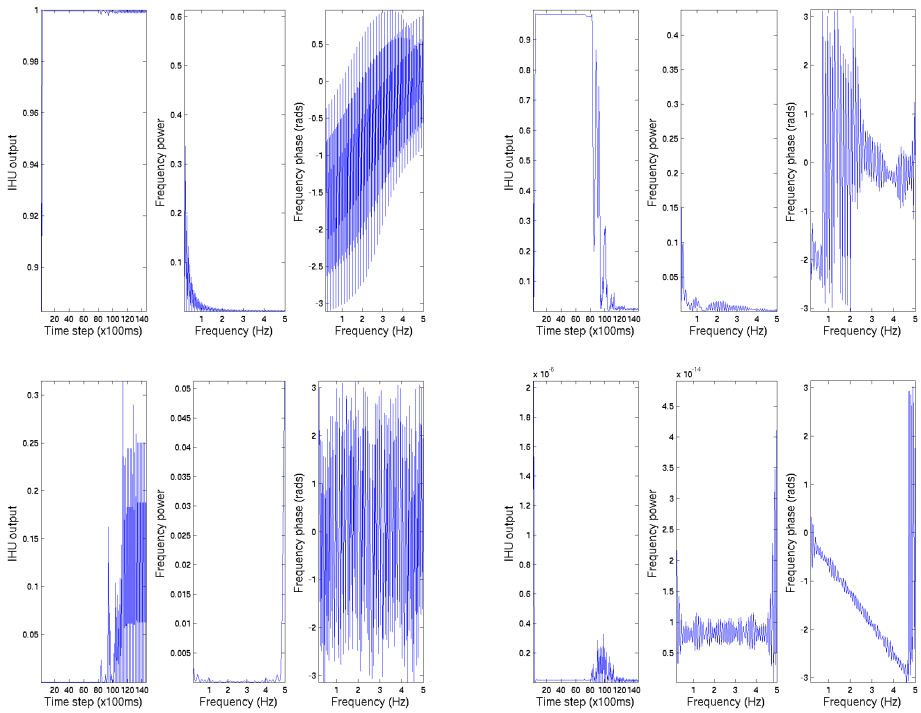


Figure F.6: Frequency plot of actuator IHUs in situation 6a. From left to right, top to down, sensor M_L, M_R, P_T, P_R.

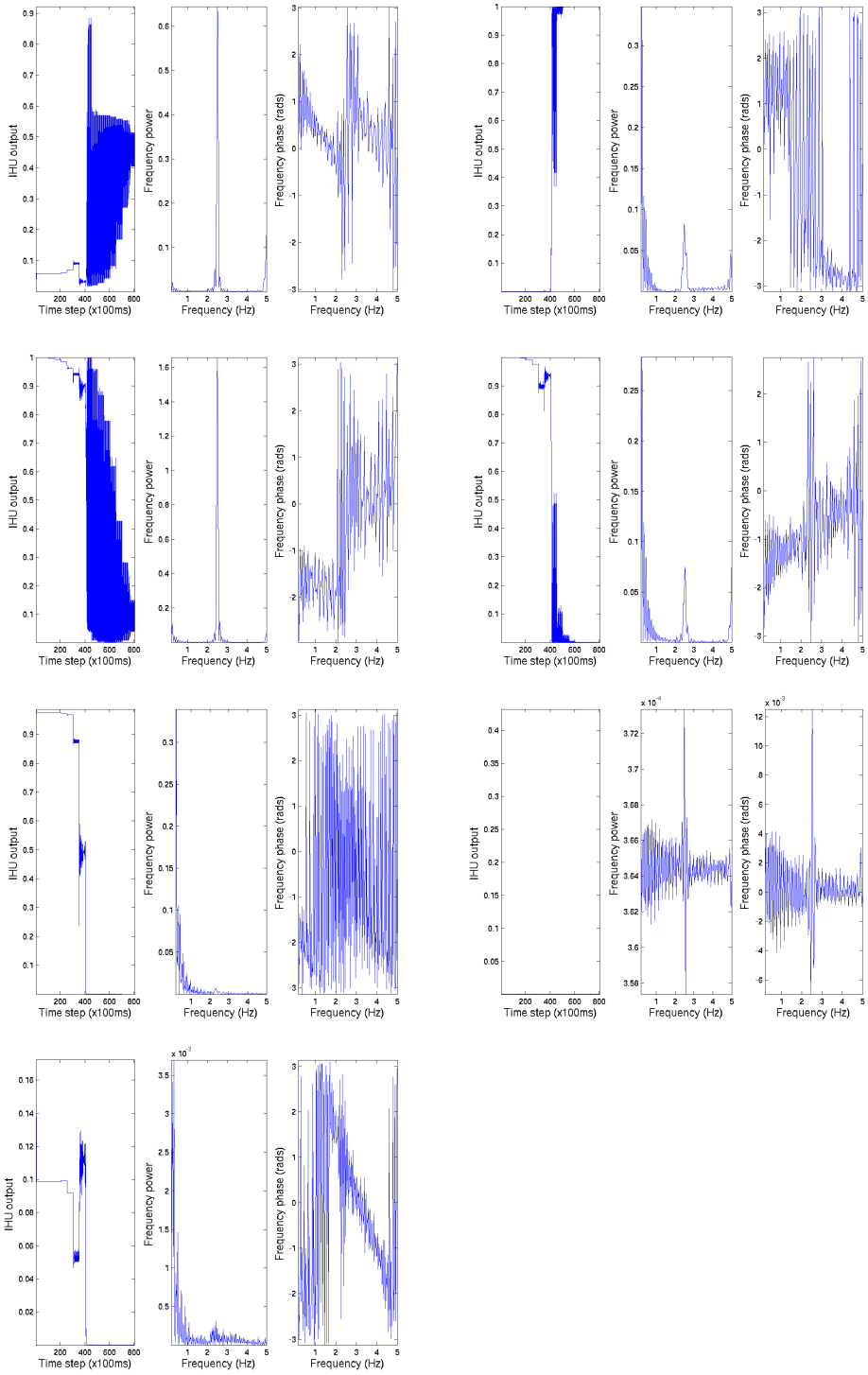


Figure F.7: Frequency plot of sensor IHUs in situation 11a. From left to right, top to down, sensor S_A, S_B, S_C, S_D, S_E, S_F, S_G.

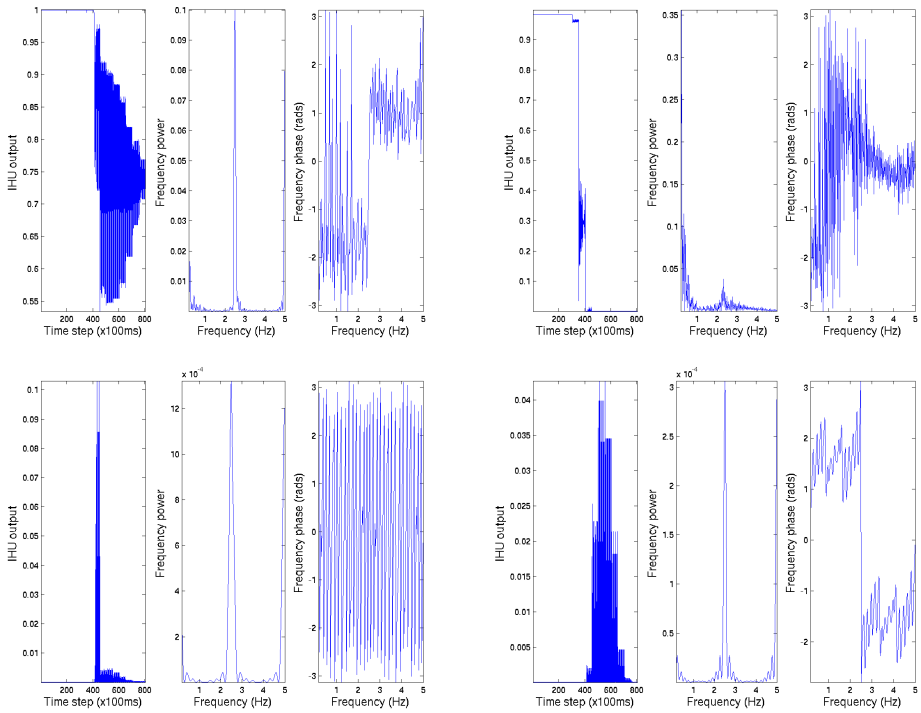


Figure F.8: Frequency plot of actuator IHUs in situation 11a. From left to right, top to down, sensor M_L, M_R, P_T, P_R.

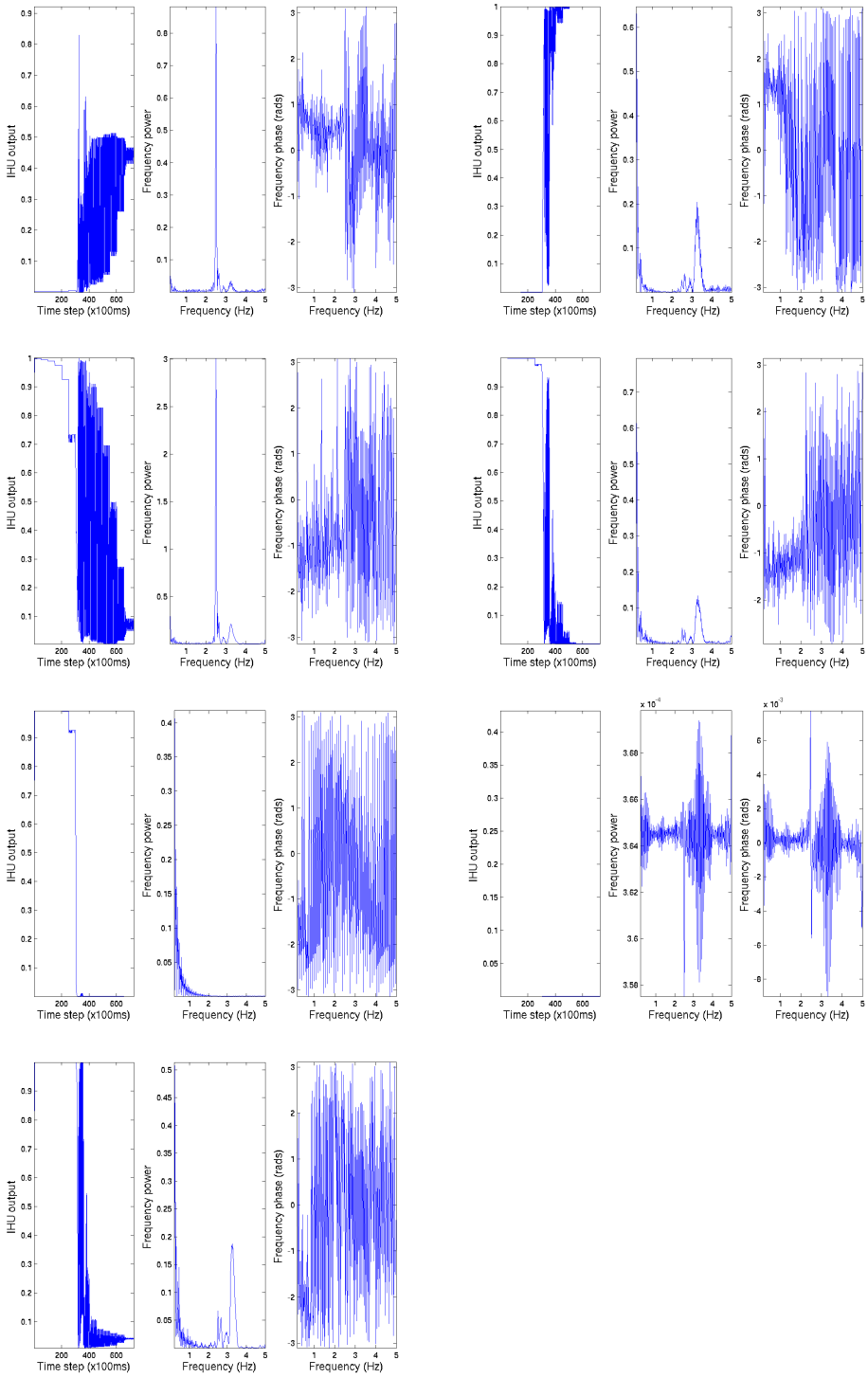


Figure F.9: Frequency plot of sensor IHUs in situation 11b. From left to right, top to down, sensor S_A, S_B, S_C, S_D, S_E, S_F, S_G.

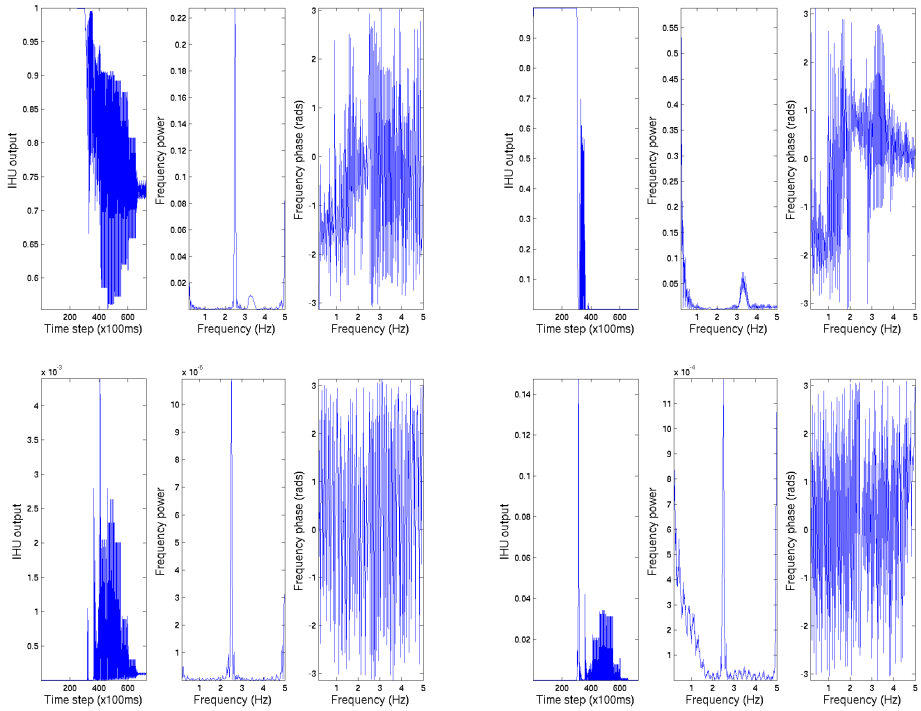


Figure F.10: Frequency plot of actuator IHUs in situation 11b. From left to right, top to down, sensor M_L, M_R, P_T, P_R.

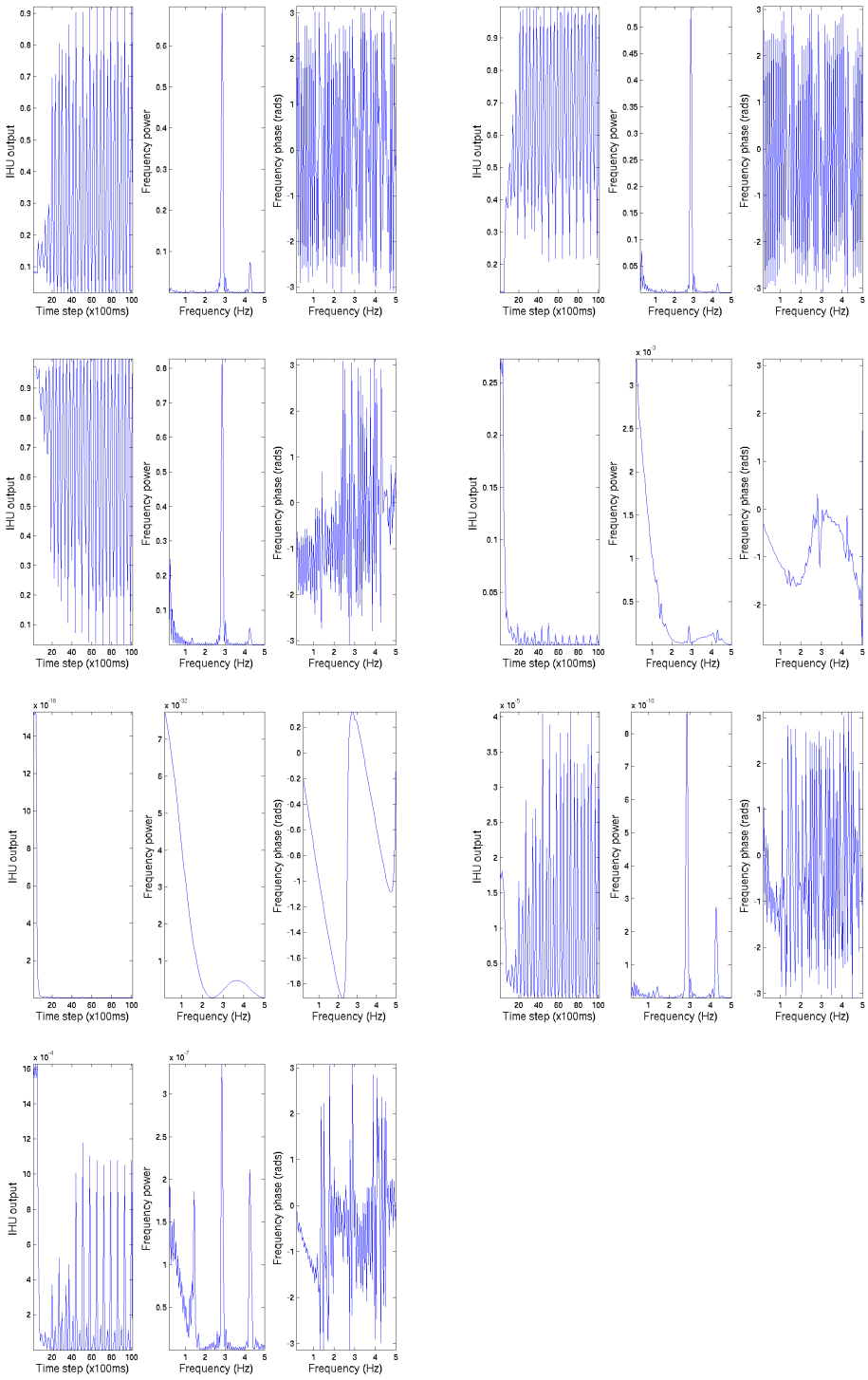


Figure F.11: Frequency plot of sensor IHUs in situation 12a. From left to right, top to down, sensor S_A, S_B, S_C, S_D, S_E, S_F, S_G.

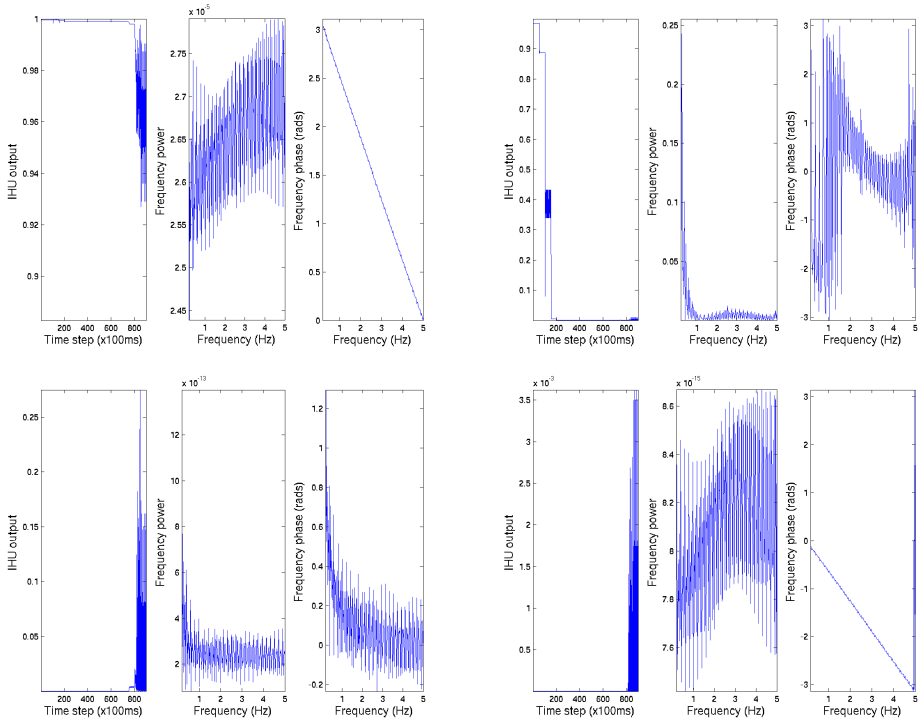


Figure F.12: Frequency plot of actuator IHUs in situation 12a. From left to right, top to down, sensor M_L, M_R, P_T, P_R.

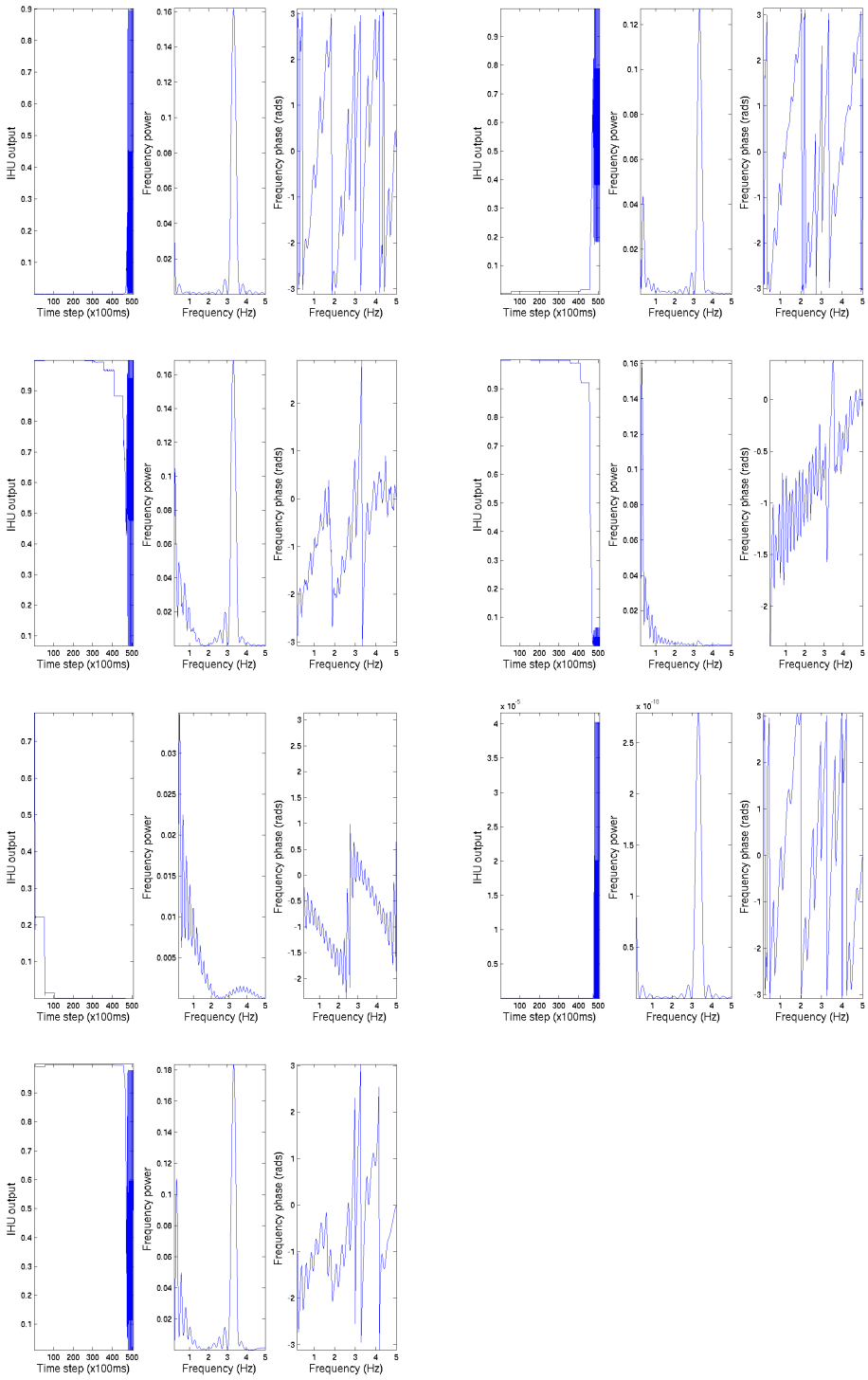


Figure F.13: Frequency plot of sensor IHUs in situation 12b. From left to right, top to down, sensor S_A, S_B, S_C, S_D, S_E, S_F, S_G.

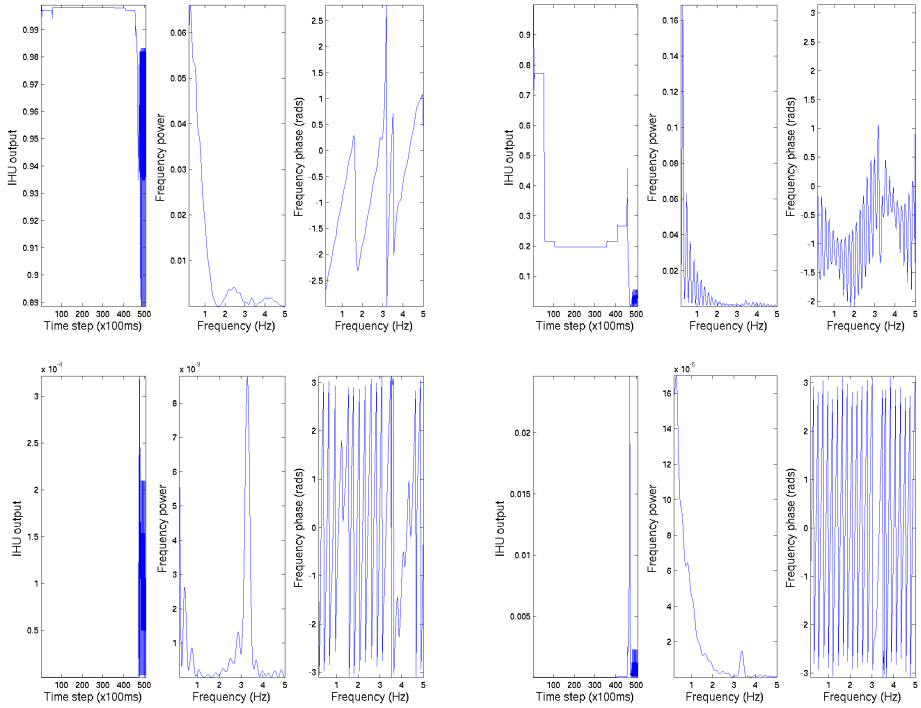


Figure F.14: Frequency plot of actuator IHUs in situation 12b. From left to right, top to down, sensor M_L, M_R, P_T, P_R.