

Autonomous Learning Through Evolutionary Robotics

Senior Design Project Report

by

Muhammad Ali Khattak

Omar Khan

Rizwan Ahmad Khan

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF BACHELOR OF SCIENCE

GIK Institute of Engineering Sciences & Technology

Faculty of Computer Science & Engineering

May 2003

Copyright © 2003 by GIK Institute.

All rights reserved. No part of this document shall be reproduced, stored in a retrieval system or transmitted by any means, electronic, mechanical photocopying, recording, or otherwise without the written permission of the G.I.K Institute.

Abstract

This project is an undertaking in the field of evolutionary robotics. It aims at the evolutionary development of a real, neural network driven mobile robot. In this project the evolutionary process takes place in a simulated environment and then is transferred to a real mobile robot. The neural controllers of the evolved best individuals display a full exploitation of non-linear and recurrent connections that make them more efficient than analogous man-designed agents. Furthermore, the focus is on the evolution of the adaptive rules for each neuron of the controller than the synaptic weights. The connections thus use these rules to adapt their strength online starting from random values, thus increasing robustness.

XINC is an AI-based application that produces a Neural Controller which would enable a robot to move autonomously. The robot would exhibit mobility in an unknown environment and avoid collision with obstacles. The evolution of the optimal Neural Controller is done using Genetic Algorithms.

Table of Contents

ACKNOWLEDGEMENTS	7
DEDICATION	8
INTRODUCTION	9
1.1 BACKGROUND	9
1.1.1 <i>Introduction to Robotics</i>	9
1.1.2 <i>Behavioral Robotics</i>	10
1.1.3 <i>Evolutionary Robotics</i>	10
1.1.4 <i>Neural Controllers</i>	10
1.2 INTRODUCTION	11
THEORY	12
2.1 EVOLUTIONARY ALGORITHMS	12
2.1.1 <i>Genetic Algorithms</i>	12
2.1.2 <i>Artificial Neural Networks</i>	13
2.1.2.1 Supervised Error-based Learning	15
2.1.2.2 Reinforcement Learning	15
2.1.2.3 Unsupervised Hebbian Learning	15
2.2 EVOLVING ARTIFICIAL NEURAL NETWORKS	16
2.3 CREATING AUTONOMOUS ROBOTS	18
2.3.1 <i>Autonomy</i>	18
2.3.2 <i>Situatedness and Embodiment</i>	19
2.3.3 <i>Adaptiveness</i>	19
2.4 EVOLUTIONARY ROBOTICS	19
2.5 PHYSICAL ROBOTS OR REALISTIC SIMULATIONS?	20
DESIGN & IMPLEMENTATION	22
3.1 REQUIRED FUNCTIONALITY	22
3.2 MODULAR BREAKDOWN	22
3.2.1 <i>Flow of control</i>	23
3.3 SUB-MODULAR BREAKDOWN:	24
3.3.1 <i>Genetic Algorithm Module:</i>	24
3.3.1.1 Population Initialization:	24
3.3.1.2 Genetic Operators:	24
3.3.1.3 Population Evolution:	28
3.3.2 <i>Neural Network Module:</i>	28
3.3.2.1 Gene Decoding:	29
3.3.3 <i>Evaluation Module:</i>	29
3.3.3.1 World Creation:	29
3.3.3.2 Robot Modelling:	30
3.3.3.3 Obstacle Detection:	30
3.3.3.4 Remote Sensor Modelling:	31
3.3.4 <i>Deployment Module:</i>	31
3.4 INFORMATION FLOW:	32
3.5 IMPLEMENTATION ISSUES:	33
3.5.1 <i>Genetic Algorithm Implementation:</i>	33

3.5.2	<i>Plastic Neural Network Implementation:</i>	33
3.5.3	<i>Rotation of Motors:</i>	33
3.5.5	<i>Obstacle Modelling:</i>	34
3.5.6	<i>Random Number Generation:</i>	34
3.5.7	<i>Parallel Execution of GA and Simulation:</i>	34
3.5.8	<i>Storage of Parameters of experiments:</i>	35
OUR APPROACH		36
4.1	EVOLUTION AND ADAPTATION:	36
4.1.1	<i>Encoding Mechanisms of Adaptation:</i>	36
4.2	TASK: OBSTACLE AVOIDANCE:-	40
4.3	ANALYSIS OF RESULTS:	41
4.4	EVOLUTION IN CHANGING ENVIRONMENTS	43
CONCLUSION		45
5.1	CONTRIBUTIONS:	45
5.2	LIMITATIONS:	46
5.3	FUTURE DIRECTIONS:	47
APPENDICES		48
A.1	CODE FOR GENERATING RANDOM WEIGHTS FOR INITIAL STARTUP OF SIMULATION:	48
A.2	CODE FOR CALCULATING NEW POSITION OF ROBOT AFTER OUTPUT HAS BEEN MAPPED ONTO MOTORS:	48
BIBLIOGRAPHY		50

List of Figures

Figure 1: Neural Network	14
Figure 2: Presynaptic and Postsynaptic Connections	16
Figure 3: Weight Encoding	17
Figure 4: Weights Equation	17
Figure 5: Schematic Representation	23
Figure 6: Flow of Control.....	23
Figure 7: Binary String Genome.....	24
Figure 8: One Point Crossover	25
Figure 9: Two Point Crossover	25
Figure 10: Mutation.....	26
Figure 11: Class of Binary String Genome	27
Figure 12 Screenshot #1 Setting Parameters	28
Figure 13: Evolve Diagram	28
Figure 14: Weight Encoding.....	29
Figure 15: Screen Shot #2 Create World.....	30
Figure 16: Viewing Angle	31
Figure 17:Information Flow	32
Figure 18: Output Mapping.....	33
Figure 19: Wall Detection.....	34
Figure 20: Parallel Execution	34
Figure 21: Encoded Features	39
Figure 22: Bit Encoding.....	39

ACKNOWLEDGEMENTS

The authors wish to express their utmost gratitude and sincere appreciation to Dr. Anwar M. Mirza and Dr. A.P. Engelbrecht for their supervision & assistance during the course of this project. In addition, special thanks to Engineer Zakaria Hadi and Alexandre Colot for their help.

DEDICATION

The authors dedicate this endeavour to their Teachers, Parents and GIKI.

Chapter 1

Introduction

“The past three centuries of science have been predominantly reductionist, attempting to break complex systems into simple parts, and those, in turn; into simpler parts. The reductionist program has been spectacularly successful, and will continue to be so. But it has often left a vacuum: How do we use the information learned about the parts to build up a theory of the whole? The deep difficulty here lies in the fact that the complex whole may exhibit properties that are not readily explained by understanding the parts. The complex whole, in a completely non-mystical way, can often exhibit collective properties, “emergent” features that are lawful in their own rights...”

— *At Home in the Universe*

By Stuart Kauffman

1.1 Background

1.1.1 Introduction to Robotics

The word *robot* was first used in 1920 by the Czech author Karel Capek, who derived it from the Czech word for slave, *robota*. In 1942, Isaac Asimov, an American science fiction writer, wrote a short story in which he introduced the word *robotics* as well as the now famous *Three Laws of Robotics*:

1. A robot may not injure a human being, or, through inaction, allow a human being to come to harm.
2. A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Today, a *robot* means a mobile computer situated in the real world interacting with the environment through sensors and actuators in order to perform various tasks. The word *robotics* is the science of building and programming robots.

Programming stationary computers and robots are two very different tasks. On stationary computers the context of the program is mostly well known and deterministic. Robot programming is very different — nothing is known, everything is a guess; as the robot knows nothing but what its sensors tell it and since we must teach the robot how to derive a representation of the world around it from this sensor image. The

amount of noise and the endless possibilities of different sensor images in a dynamic environment, make robot programming very difficult.

1.1.2 Behavioral Robotics

In the mid-eighties, a new paradigm of robotics emerged, Behavioral Robotics (BR). It was pioneered by Rodney Brooks in 1986 as a reaction to the classical view of robotics. In Behavioral Robotics, the robot control is decentralized and divided into different behaviors running in parallel. Instead of the task decomposition, the method is behavioral decomposition. Brooks showed that his subsumption architecture was capable of outdoing everything seen until then in terms of robustness and reactivity.

1.1.3 Evolutionary Robotics

The idea of Evolutionary Robotics was hinted at in Braitenberg's *Vehicles*, where it is suggested to use a sort of Evolutionary Algorithm for building robots. Rather than attempting to hand-design a system to perform a particular task or range of tasks well, the evolutionary approach will allow a gradual emergence of the sought after behavior.

Though more closely related to Behavioral Robotics than to Classic Robotics, Evolutionary Robotics is a third way of robotics; it uses neither *functional decomposition* nor *behavioral decomposition*. Given a suitable control structure (brain), it is utterly up to evolution to figure out the best way to solve the task. It is possible for the designer to split a large task into subtasks and evolve these.

The motivation for Evolutionary Robotics is that interesting robots are too difficult to design by hand. In Evolutionary Robotics (ER), the Evolutionary Algorithm (EA) is used to evolve autonomous robots for a specific task, and thereby evading the difficulties of hand-designing autonomous robots. Basically, the process is to start with a generation of robots: Test them all, select the best for breeding, perform some recombination and mutation to produce the next generation, and continue until a satisfactory robot is found.

The Evolutionary Algorithm is a well suited optimization method for robotics, and although EA is generally not good at finding optimal solutions, it is, if properly used, known for finding robust solutions — and *robustness* is one of the most sought-after qualities in robotics.

1.1.4 Neural Controllers

This aspect of our project shall be elaborated upon further in another section of this report. In a nutshell, neural controllers are a class of feed-back and self-organizing control systems that are based on Artificial Neural Networks. Their behavior i.e., learning and operation is normally categorized in the un-supervised knowledge acquisition systems.

1.2 Introduction

XINC consists of a mobile vehicle capable of autonomous learning (self-learning) in an unknown terrain along with its software. It tries to navigate in the terrain while avoiding any obstacles it might encounter.

This Project IS an effort to study the theory behind concepts such as ANN's and GA's and their subsequent combination to form neural controllers which exhibit evolving intelligence with successive generations. The objective is to develop, initially in simulation and later, in a practical application a neural controller capable of learning in an evolving fashion.

This project IS NOT an effort to optimize previously discussed concepts such as path-finding and terrain mapping. It will not address the details of robot mechanics etc.

The whole system comprises of a number of interconnected modules. Initially, the Neural Controller is generated by a stand-alone application based on GA's. The resultant ANN controller is transferred onto the memory of a *Hemisson* robot (*Hemisson*, and its details will be addressed subsequently). Once the robot acquires its 'brain', it is capable of exhibiting autonomous/intelligent behaviour. The application is designed and implemented to be executed on a stand-alone desktop PC with sufficient processing capability (>700 MHz).

Chapter 2

Theory

“Whence arises all that order and beauty we see in the world?”

— *Principia Mathematica*

By Isaac Newton

2.1 Evolutionary Algorithms

2.1.1.1 Genetic Algorithms

Genetic Algorithms were first introduced by Holland (1975) as a parallel search technique based on the Darwinian principle of selective reproduction of fittest individuals.

A genetic algorithm operates on a *population* of artificial chromosomes (genotypes), which is a string that encodes the properties of a population of individuals (phenotypes). Encoded properties represent some variables of the individual such as the synaptic weights of a neural network. An artificial chromosome can be thought of as the individual’s D.N.A., composed of a number of *genes*, each one containing a symbol or *allele* from a set of possible symbols. Although several kind of encoding methods have been used, the most common one consists of encoding each gene using a set composed of two alleles, 0 and 1. This method is known as *binary encoding*.

A genetic algorithm starts with an initial population of random chromosomes (in the case of binary genotypes, this corresponds to random strings of 0’s and 1’s). Each chromosome is decoded into the corresponding phenotype, one at a time, and the performance of the phenotype is evaluated. The performance of the individual corresponding to a given genotype is referred to as *fitness*, and the function that computes this performance is known as *fitness function*. Then genetic operators –selective reproduction, crossover, and mutation – are applied to create a new generation of chromosomes. This evolutionary process is repeated for several generations until a good individual is found or until the best fitness in the population stops increasing. It is important to distinguish between genotype and phenotype, and to notice that while genetic operators are applied on the genotypes, fitness is measured on the phenotypes.

Selective reproduction consists of making copies of the chromosomes in the population, with a probability proportional to the fitness of the individual. Therefore fittest individuals will have more chances of reproduction than individuals that scored a poor performance. Several methods implement the selective reproduction operator:

- 1- *Roulette-wheel* method consists of a wheel in which each individual is represented as a slot. The width of the slot is proportional to the fitness of the individual it represents. Therefore, since copies are made by spinning the wheel and selecting the individual corresponding to the slot where the ball stops, individuals with higher fitness have more reproduction chances.
- 2- *Rank-based* works by selecting individuals from a ranking in which they are classified from best to worst.
- 3- *Truncation selection* is another rank-based method. The N individuals of a population are ranked from best to worst and the best M are selected to make K copies such that $MK = N$.
- 4- *Tournament selection* is based on the competition of two individuals randomly selected from a population. If a random number between 0 and 1 is smaller than a predefined parameter T , the individual with the highest fitness is selected otherwise the less performant individual will be copied to the next generation.

After the selection process, crossover and mutation operators are applied with a given probability on reproduced individuals. *Crossover* consists of taking two chromosomes, selecting a random point along them, and swapping their genes around that point. This recombination mechanism aims combining in a unique genotype potential building blocks that could eventually provide selection advantage. *Mutation* introduces new genetic material by switching randomly selected genes on the phenotype. In some cases, the best individual of a generation is copied to the next generation without modifications. This mechanism is known as *elitism* and aims at preserving the best solution so far.

2.1.2 Artificial Neural Networks

The control system of an autonomous robot is often implemented as an artificial neural network. There are several reasons for this choice:

1. Neural networks provide a straightforward mapping between sensors and motors, and can handle continuous or discrete input and output signals.
2. Neural networks are robust to noise, since their units are based upon a sum of several weighted signals, and oscillations in individual values of these signals do not drastically affect the behaviour of the network. This property is very important in the context of the robots with noisy sensors that interact with noisy environments.
3. Neural networks can be trained using different types of learning algorithms.
4. Neural networks provide various levels of granularity. One may apply evolutionary or learning techniques to the lowest level specification of a neural network, such as the connection weights, or to higher levels, such as the coordination of predefined modules composed of predefined sub-networks.

Artificial neural networks are parallel computing structures crudely inspired by biological nervous systems. They are composed of a set of basic processing unit, or *nodes*, connected by weighted links called *synapses*.

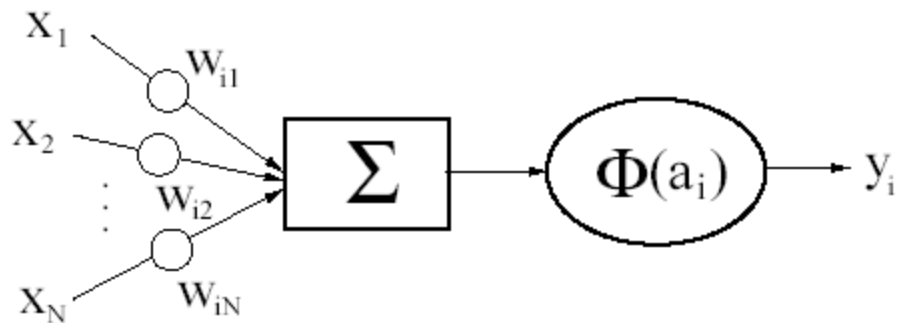
Artificial neural networks can be organized in different kind of architectures. Figure shows on the left side a conventional *feed-forward* architecture which consists of one or more *input units*, which receive information from the environment, of zero or more *internal units*, and of one or more *output units*, which send information to the environment. On the right side, a *recurrent* neural network is represented. It also has input, hidden, and output units, but in this case there is no clear direction for the information flow, since there are many feed-forward connections between neurons, including self-connections. Recurrent architectures present more complex time-dependent dynamics than feed-forward neural networks.

The most commonly used model of an artificial neuron is shown in the figure. It computes the output y_i of the unit i as a function $\Phi(a_i)$ of its activation a_i , which is calculated by adding all incoming inputs x_j weighted by the connection strengths w_{ij} .

$$y_i = \Phi\left(\sum_{j=1}^N w_{ij}x_j\right)$$

Several kinds of transfer functions can be used to compute the output of the neuron. Commonly used ones include the step function, the linear function, and the sigmoid function.

Figure 1: Neural Network



Artificial neural networks can compute a vast amount of information by transmitting many signals in parallel through the connections between neurons. Nevertheless, obtaining a desired behavior by setting the strength of synapses is not at all an easy task, and can become even impossible in the case of large networks. Fortunately, several learning techniques have been conceived to allow neural networks to automatically adjust synaptic strengths in order to produce specific input-output mappings. In *supervised error-based learning*, for example, the modifications of the synaptic weights of the network is a function of the error between the output of the network and the desired output. This implies that desired output must be known for each input pattern. Instead, *reinforcement learning* algorithms exploit high level supervision signals instead of explicit desired outputs for each input pattern. *Unsupervised Hebbian learning* does not need any supervision signals, and relies rather on self-organization algorithms.

2.1.2.1 Supervised Error-based Learning

Determining by hand the strength of synaptic weights to generate a given behavior is not always evident, especially as the size of the network increases. However, in some cases, the designer knows the desired output of the system for some input patterns. This is the case, for example, of hand-written text recognition or of Text to Speech Conversion problem, where the correct output for a given input is known. In this case, *supervised error-based learning* can be applied by using the error between the output and the desired output to adjust the synaptic weights, so that error is minimized. Once that network has learned the correct output for the training input patterns, it can produce correct output responses also for input patterns that were not included in the training data. This property is also known as *generalization*.

2.1.2.2 Reinforcement Learning

In some cases, the desired output of the network can be defined only in a rough way or even is not available continuously at every learning step. This is usually the case for autonomous mobile robots. The feedback obtained from the environment is often related to the global behavior of the robot, but is not at all a detailed description of the desired output. In other words, feedback received by the robot is a kind of performance measure, but it does not explicitly determine the correct output.

Reinforcement Learning aims at exploiting this kind of rough and discontinuous information by generating the input-output mapping that maximizes the positive reward received by the agent over time.

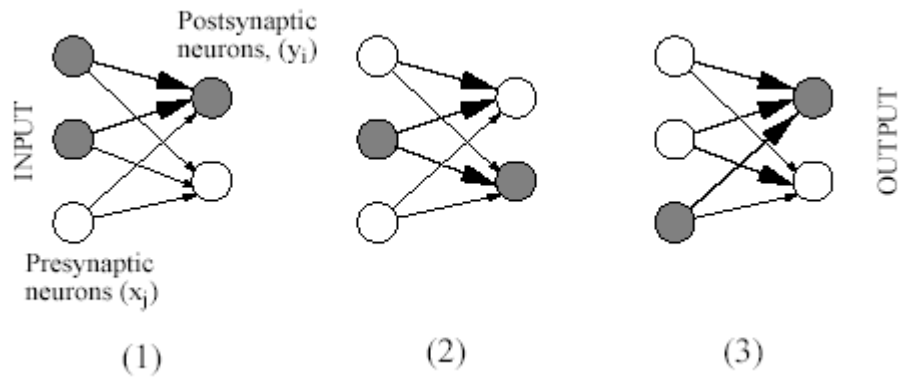
2.1.2.3 Unsupervised Hebbian Learning

In supervised learning algorithms, such as Back-propagation or reinforcement learning, behavior is mainly shaped by the training data or the performance criterion provided by the designer. Instead of using this kind of external knowledge, which is not always easy to define, other algorithms aim at exploiting self-organization to extract information from the interactions between the agent and the environment. Unsupervised Hebbian learning, for example, does not need any desired output or reinforcement signals to learn a given input-output mapping. Instead, the input patterns, the learning rule, and the architecture determine how the network *self-organizes* its behavior.

In 1949, a Canadian psychologist Donald Hebb hypothesized that the nervous system functions by strengthening the connection between two neurons that are simultaneously active:

$$\Delta w_{ij} = y_i x_j,$$

where x and y are the activations of the presynaptic and postsynaptic neurons, respectively. In Hebbian learning input patterns are presented to the network, one at a time, and after each presentation the synapses connecting two neurons that are simultaneously active are strengthened (figure). After learning, this kind of network is capable of providing the correct output even for incomplete version of the input pattern is presented.

Figure 2: Presynaptic and Postsynaptic Connections

2.2 Evolving Artificial Neural Networks

Many approaches to the evolution of control systems for autonomous robots described in the literature resort to artificial neural networks to construct their control architectures. In fact, neural networks offer a rich variety of possibilities to apply evolutionary approaches. From the strength of the synaptic weights to the architecture of the network, many different aspects can be included in the evolutionary process.

Moreover, neural networks allow for different levels of adaptation such as phylogenetic adaptation (evolution), development (maturation), and ontogenetic adaptation (lifetime learning), which can be combined in various ways.

Some of the approaches to the evolution of neural networks are given below.

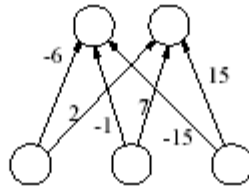
Synaptic Weights

A typical approach to the evolution of neural networks consists of encoding the synaptic weights of the network in order to generate the desired input-output mapping. Evolution of synaptic weights offers several advantages with respect to other learning algorithms

1. A genetic algorithm explores a *population* of networks, not a single network;
2. There are no constraints on the type of architecture, output function, or other parameters;
3. Desired output for each input pattern is not required as in supervised learning methods.

The method consists of directly encoding the synaptic weights as a string of binary values with a given precision, or as a string of real values (Figure).

Figure 3: Weight Encoding



(a)

10110	00010	10001	00111	11111	01111
-------	-------	-------	-------	-------	-------

(b)

-6.0	2.0	-1.0	7.0	-15.0	15.0
------	-----	------	-----	-------	------

Performance of a genetic algorithm has been compared with that of back-propagation algorithm by Montana and Davis (1989) in different classification tasks. The results showed that evolution is capable of generating more performant solutions in less computational cycles.

Architectures

The choice of an appropriate architecture is crucial to construct performant neural networks, but they are difficult to design by hand. Therefore, genetic algorithms are sometimes used to automatically evolve neural morphologies. However, encoding every detail of the architecture may generate extremely long genotypes, which are difficult to evolve. For this reason, typically only a few construction parameters are encoded to generate more or less complex network architectures from one-dimensional genotypes.

Learning Rules

Genetic algorithms have been also employed to evolve learning rules for neural networks. This method was proposed by Chalmers (1990). He proposed to encode synaptic weight changes as a function of only information local to the synapse, and to use this same function to change every synaptic weight of a single-layer feed-forward neural network::

Figure 4: Weights Equation

$$\Delta w_{ij} = k_0(k_1 w_{ij} + k_2 a_j + k_3 o_i + k_4 t_i + k_5 w_{ij} a_i + k_6 w_{ij} o_i + k_7 w_{ij} t_i + k_8 a_j o_i + k_9 a_j t_i + k_{10} o_i t_i),$$

where a_j is the activation of the input nit j , and o_j the activation of the output unit i , t_i the training signal on the output unit i , w_{ij} the current value of the synaptic strength for the input j to output I , and $k_c, 0 < c \leq 10$ are the parameters to be evolved.

Chalmers evolved a population of learning individuals on a set of supervised numerical tasks, where each individual is presented a number of training signals for each task. All the synaptic weights of the network corresponding to the individual are randomly initialized at the beginning of individual's life, and evolved learning function is used to change synaptic weights during individual's life. The results

showed that some of the individuals rediscovered the Delta Rule, and that evolution in a diverse and unpredictable environment can produce adaptive mechanisms capable of coping with a variety of different environmental features as they arise.

Floreano and Mondada (1996) described an experiment in which strength of synaptic weights is not directly encoded into the genotype, but is continuously modified during lifetime through a learning process in which genetically-determined adaptation rules interact with the information coming from the external environment, without using any training data. They employed a neural network to control a mobile robot on a simple navigation task. The genotype encodes for the properties (not the strength) of each synapse of the network: driving or modulatory, excitatory or inhibitory, learning rate, and learning rule. Each synapse can change according to one of the four Hebbian learning rules: Plain Hebbian, presynaptic, post synaptic, or covariance. Synaptic weights are randomly initialized and the final weights are not coded back into the genotype.

Robots were evolved in a real environment and selected for their ability to navigate as fast as possible while keeping far from obstacles. The results showed that the robots do not show a good navigation ability at birth (initial synaptic weight are random), but this is acquired during the initial motor loops while the robot moves in the environment. In the words of the authors, one of the most remarkable result is that the synapses continuously change while the behavior of the robot becomes rather stable after a few seconds. The synapses evolved in this experiment are responsible for both learning and behavior regulation. Knowledge in the network is not expressed by a final synaptic configuration, but rather by a *dynamic equilibrium point* in a n-dimensional state-space (where n is the number of synapses).

The results reported by Floreano and Mondada (1996) clearly show that the environment plays a crucial role in shaping the ontogenetically developed behavior. Behavior is an emergent property, and of the amount of information encoded in the genotype may be reduced given that part of the information will be provided by the interaction between inherited instructions and the environment.

2.3 Creating Autonomous Robots

Autonomous robots are identified by their ability to adapt to an environment by developing a suitable control system capable of coping with unpredictable sources of changes. A vast number of approaches, with more or less success, have been proposed to design physical and simulated robots displaying these characteristics.

Some of the concepts related to the creation of autonomous agents are given below:

2.3.1 Autonomy

Autonomy can be defined as *the ability of being self-governing*. Autonomous systems develop for themselves the laws and strategies according to which they regulate their behavior, that is they are self-regulating as well as self-governing.

Self-governing is crucial in the case of *mobile* autonomous robots. Unlike robotic manipulators (e.g. assembling arm-robots), mobile robots often operate in open, unpredictable, and dynamic environments. In this scenario, one cannot program a robot to move along a given path, but instead, has to give the robot the possibility of making decisions depending on the conditions encountered in the surrounding environment.

2.3.2 Situatedness and Embodiment

These two concepts were introduced by Brooks (1991) to stress the importance of the coupling between action and perception. *Situatedness* asserts that intelligent agents must be situated in the real world and that its behavior must be shaped by the interaction generated through its sensory-motor system with the surrounding environment.

Embodiment implies that an intelligent agent must have a physical body grounded in the world. A robot must have a physical body in order to interact with its environment. Consequently, a robot is continuously subjected to physical forces, to energy dissipation, or to any other influence in the environment.

Both situatedness and embodiment highlight the importance of the interaction with the environment. However, in many cases this aspect is not considered in robotics research, because it is very difficult to design by hand a system capable of exploiting the sensory-motor interactions with the environment. The reason is that every action performed by a robot that interacts with its environment determines at the same time the next sensory pattern that the robot will receive, which in turn will determine the next action performed by the robot.

2.3.3 Adaptiveness

An autonomous robot that operates in the real world needs to be adaptive, that is it must be capable of coping with unpredictable sources of change. To this end, an autonomous robot may be provided with adaptation mechanism, such as online learning or self organization algorithms capable of modifying its behavior. In general, any modification related to the “survival” of the robot is considered as adaptive.

2.4 Evolutionary Robotics

Evolutionary robotics is a method for the automatic creation of autonomous robots. It is inspired by the Darwinian principle of selective reproduction of the fittest individuals and relies on the interaction between the robot and its environment to automatically develop autonomous systems without human intervention.

The basic idea behind Evolutionary Robotics consists in encoding the control system of the robot on an artificial chromosome. An initial population of randomly initialized chromosomes is evaluated, one at a time, on the robot, which is free to act in the environment accordingly to the genetically specified properties. The performance of the robot is measured while it acts in the environment, and then the fittest robots are allowed to reproduce by generating copies of their chromosomes with changes introduced by some genetic operators such as crossover and mutation. The process is repeated for a number of generations until an individual displays a satisfactory behavior.

Although performance evaluation plays a major role in the evolutionary process, there is no formal principle for designing a fitness function. In fact, slight differences in the fitness function may produce completely different results, and consequently, it is not possible to compare the results obtained using two different evaluation criteria. The choice of an appropriate fitness function is often made using a trial and error process, which may be a time consuming procedure when evolving physical robots.

The motivation of using an evolutionary approach is inversely proportional to the degree of knowledge provided in the fitness evaluation. The more information is given in the fitness, the less probable the emergence of unexpected behaviors from the interactions between the robot and its environment will be. In general the fitness function should be:

1. *Implicit*: It should include a minimum of variables and constraints. For example, an implicit fitness would select robots capable of keeping their energy level beyond a given threshold. An explicit fitness function, instead, would include, for example, the distance between the robot and the energy source, the distance between the robot and the obstacles of the environment, and the speed of the robot.
2. *Internal*: Fitness should be computed using information available to the agent. For example infrared sensor activation is internal information while the absolute position of the robot in the environment is external information.
3. *Behavioral*: Fitness should be related to the behavior of the robot, and not the functional aspects of the actions needed to generate such behavior. For example, in the case of light-approaching task, a behavioral fitness would be proportional to the time spent by the robot near the light. Instead, a functional fitness would be proportional to the distance between the light and the robot.

The nature of the fitness function depends on the goal of the evolutionary run. If the goal is that of optimizing a set of parameters for a very complex (but well defined) problem in a controlled environment, then the fitness function may tend to be functional, explicit, and external. Instead, if the goal is to evolve autonomous robots capable of coping with unknown and unpredictable environments without human intervention, the fitness function should be behavioral, implicit, and internal. This kind of fitness functions allows the evolutionary process to choose and adapt its own strategy to the characteristics of the environment.

2.5 Physical Robots or Realistic Simulations?

Different strategies can be adopted to evolve control systems for autonomous robot. For example, one may apply artificial evolution directly on the physical robot, or resort to realistic simulations to evolve the control system, which is then transferred to the physical robot. The use of simulations can significantly speed up artificial evolution, which may take considerable time when several individuals must be tested on the same physical for several generations. Moreover, simulated robots do not need to deal with aspects such as energy supply and mechanical robustness. However, it is very difficult to take into account every dynamics of the real world, and consequently, controllers that work correctly in simulation may fail when

tested on the physical robot. Therefore, although the evolutionary process on the physical robot, when feasible, is preferable, simulations *validated on physical robots* may be a justifiable alternative under some circumstances. To this end several aspects should be considered:

1. Different physical sensors and actuators may (and will) perform differently because of slight differences in their electronics and mechanics. For example, two (in principle identical) infrared sensors may provide slightly different measures when placed at the same position in front of the wall. This implies that two controllers may display different behaviors in identical conditions. As a solution to this problem, Miglino et al (1996) proposed to sample and record the sensory activations of the physical robot in all possible positions in which the robot can face an obstacle, and use this information in order to compute sensory activation in simulation. However, this solution is feasible only in environments containing specific kind of objects. In fact, while perception of objects such as cylinders is only a function of the orientation and distance of the robot with respect to the object, for other objects such as rectangular box, perception is also a function of the local shape of the object.
2. Physical sensors and actuators are characterized by uncertainty. Sensors return fuzzy approximations of real measures and actuators have effects that may vary in time depending on several unpredictable causes. This problem may be relieved by introducing noise at all levels of the simulation. Several methods have been proposed for this purpose. Random values selected in a certain range can be added at each cycle of individual's life, noise can be added as random values proportional to measures returned by the sensors, or certain parameters of the simulation can be modified at different epochs.
3. The morphology of the robot and the characteristics of the environment should be accurately reproduced in the simulation in order to exploit at maximum the interactions between the robot and the environment. This implies that one should not use grid worlds for evolving control systems for the robots. Nevertheless, it has been recently argued that it is not necessary to reproduce every characteristic of the robot and the environment, but only a set of characteristics relevant to the emergence of the desired behavior. This approach is known as *Minimal Simulations*.
4. The last aspect is related to the controller itself. One way of evolving robust controllers capable of coping with the variations in the environment is to increase the adaptiveness of the controller. In fact, controllers that implement direct mappings between sensors and actuators (i.e., direct encoding of synaptic weights) are much more vulnerable to environmental variations than controllers that evolve some kind of adaptation mechanism.

Chapter 3

Design & Implementation

In order to design and implement a GA-based application that would eventually be the means to control a physical robot, several aspects of the whole system had to be taken into account. The resulting design exhibited modularity right down to the lowest levels and each module was highly coupled.

Before elaborating on the design aspects of the system, a brief overview of the required functionality is given below:

3.1 Required Functionality

The functional requirements of the system are as follows:

- a. The system would simulate a robot in a given environment to reveal parameters of interest for the objective (obstacle avoidance).
- b. The system shall initially generate a neural controller.
 - i. This neural controller will be a Neural Network obtained from the Evolution of the rules for adapting weights for a designated architecture.
 - ii. It will map a certain set of inputs (source: sensors on the robot) to a set of outputs (actuators: motors on the robot).
- c. In simulation, the robot would start “learning” in a given environment autonomously.
- d. Successive reinforcement control and learning would be preformed through the simulation.
- e. The neural controller with the desired behaviour will be transferred onto a mobile robot using a hardware interfacing module.
 - i. The size of the ANN would be kept under a constraint of being smaller than the available memory of the actual robot.

3.2 Modular Breakdown

The fundamental modules of the system are as follows:

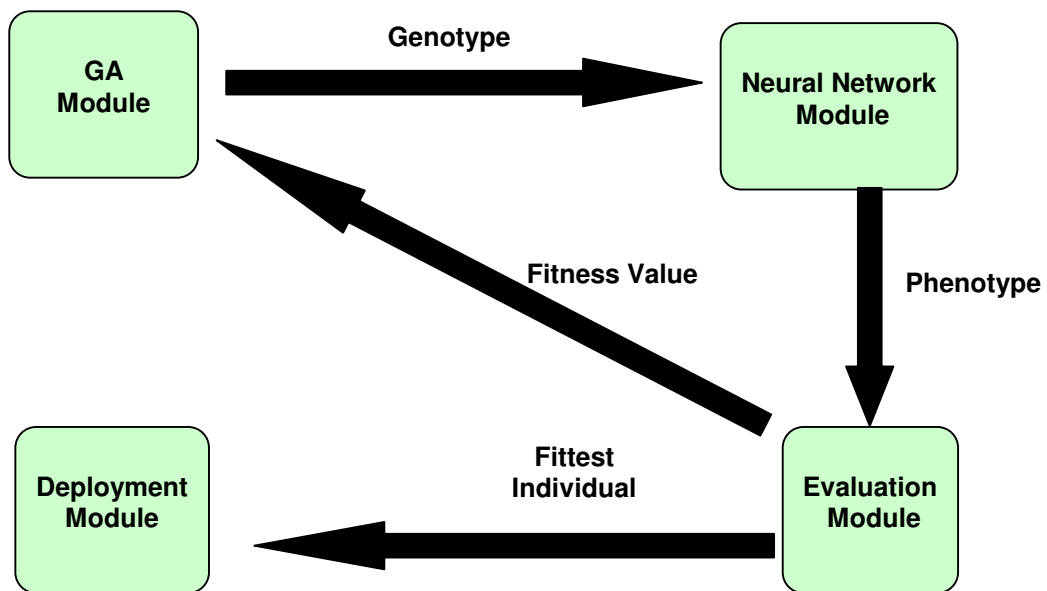
1. GA Module

2. Neural Network Module
3. Evaluation Module
4. Deployment Module

Each module is highly cohesive and consists of numerous sub-modules (to be discussed later).

The schematic representation of the whole system is as follows:

Figure 5: Schematic Representation



3.2.1 Flow of control

In terms of operation, the GA Module is the centric portion of the system as it initiates and terminates the entire process of evolving the required neural controller. The next two modules that come into execution are the neural network and evaluation module (simultaneously). The flow of control is graphically depicted as follows:

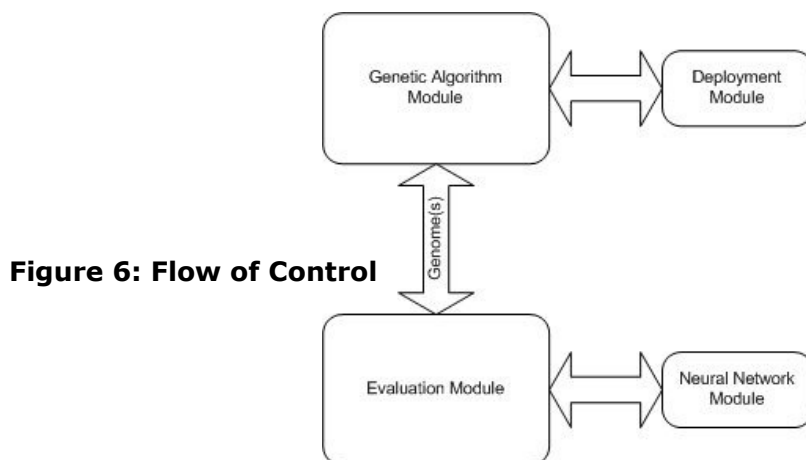


Figure 6: Flow of Control

3.3 Sub-Modular Breakdown:

3.3.1 Genetic Algorithm Module:

The GA module is one of the core modules of the system and is responsible for the generation and evaluation of the individuals for each generation. This module incorporates the basic implementation of a genetic algorithm that employs various population modification parameters such as mutation, two-way cross-over and hall of fame etc.

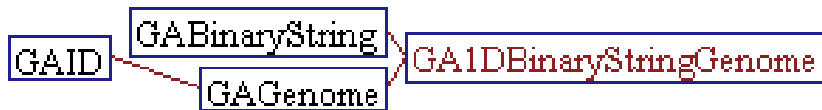


Figure 7: Binary String Genome

The above shown figure represents the base classes of the chromosome.

The design of the GA Module is broken down into the following sub-modules:

3.3.1.1 Population Initialization:

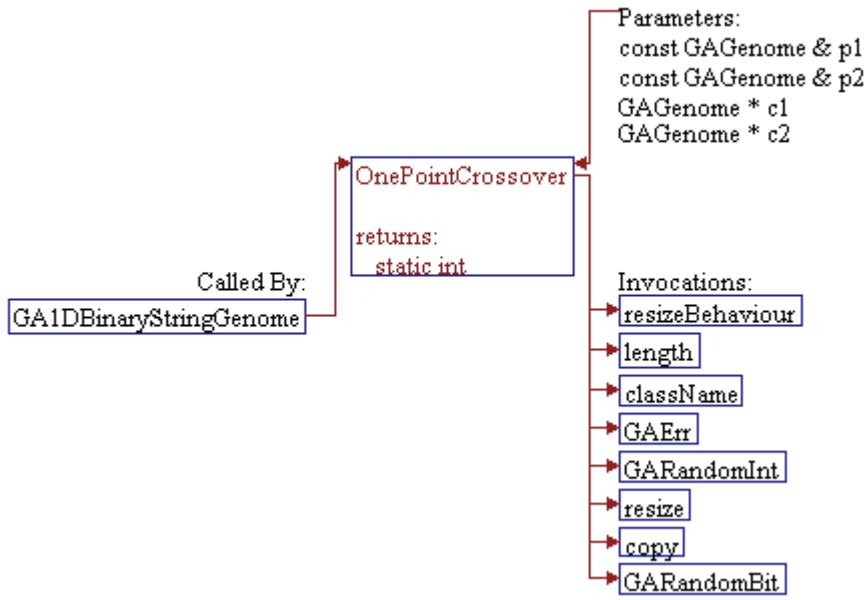
This sub module is concerned with initiating the random number generation module (provided by the base operating system) and using it to generate a specific population, tailored to the parameters given by the user. Once the population is initialized each individual is passed on to the actual algorithm in a data structure.

3.3.1.2 Genetic Operators:

Once each individual of a generation has been evaluated, the ones with the best behavior are selected and specific genetic operators are applied to them to generate the next population. These genetic operators are a major focus of the research study of project *XINC* and the currently designed set of operators are as follows:

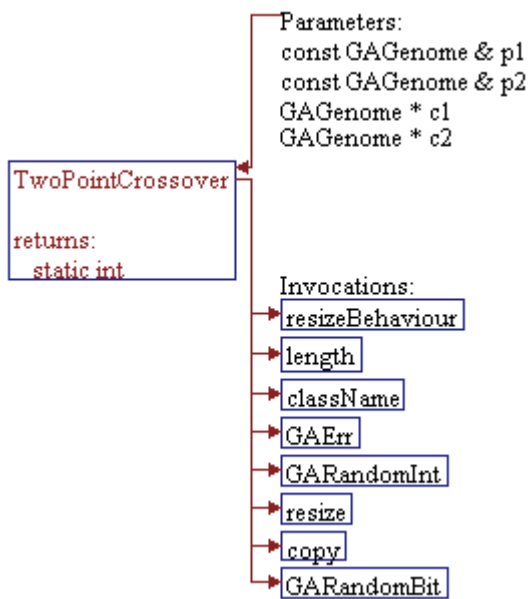
- One point cross-over.

Figure 8: One Point Crossover



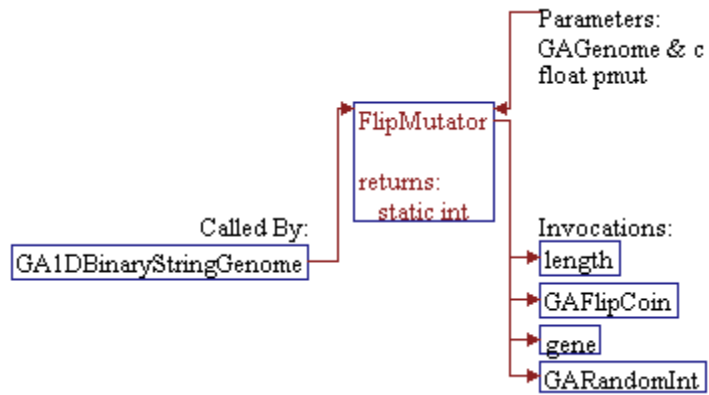
- Two point cross-over.

Figure 9: Two Point Crossover



- Mutation.

Figure 10: Mutation

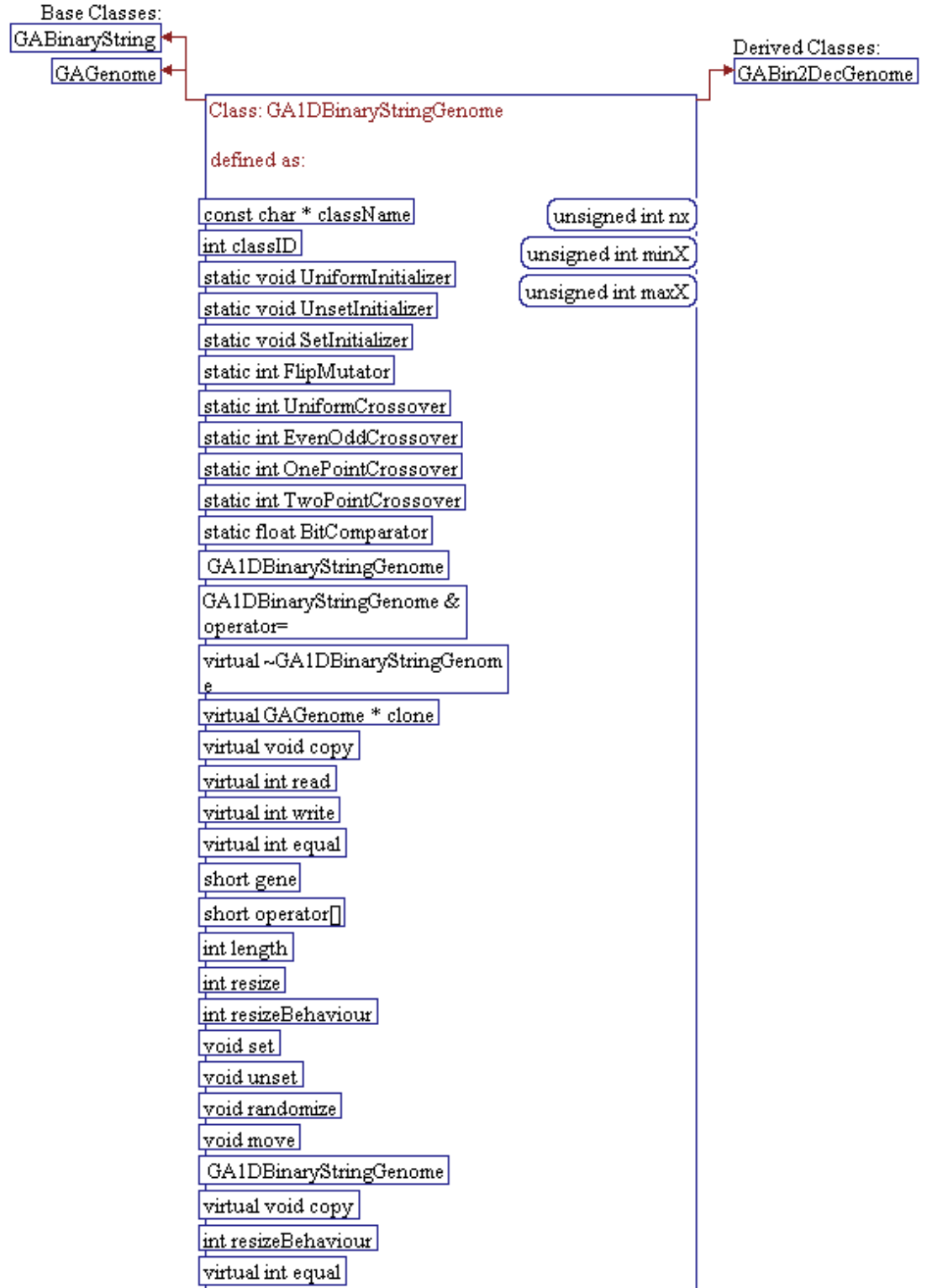


- Hall of Fame.
- Elitism.
- Random Individual.

Further details of this module are extensively linked with the form of implementation. Since this module is based on a freely available library (GAlib) for genetic algorithm components, the class diagrams etc are representing features of the GAlib C++ library.

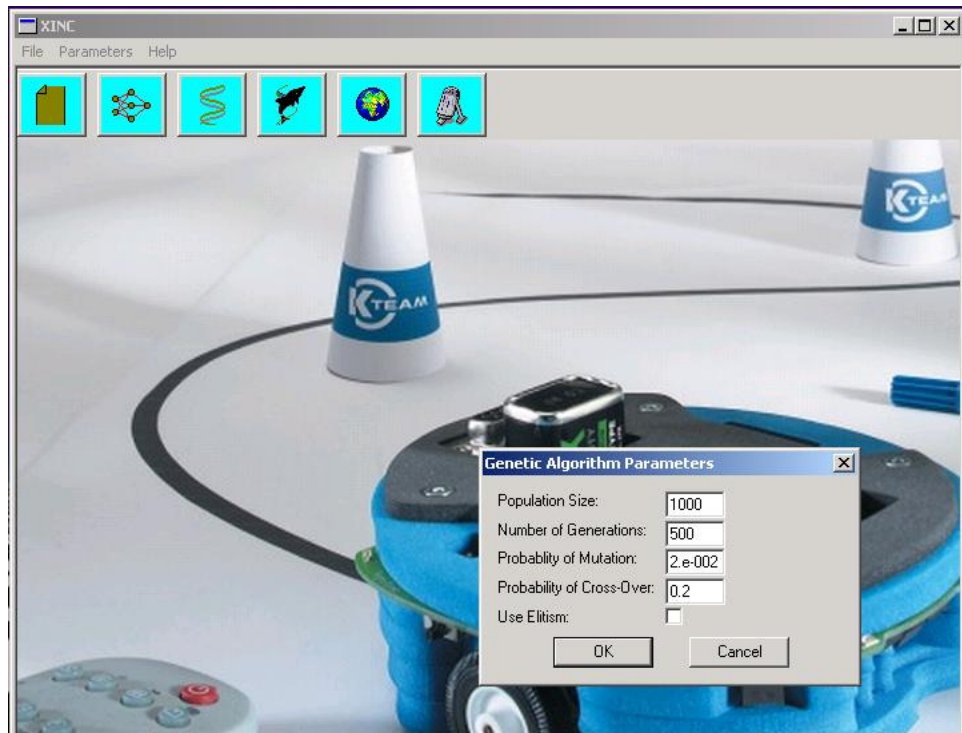
The following is a top level diagram of the operations available in GALib for a one dimensional binary string genome:

Figure 11: Class of Binary String Genome



The options to set up during the usage of *XINC* are available as shown in the following screen shot:

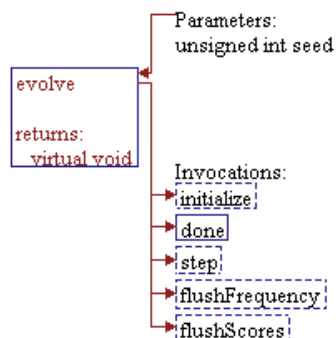
Figure 12 Screenshot #1 Setting Parameters



3.3.1.3 Population Evolution:

This module is heavily coupled with the Evaluation module. Its primary task is to apply the genetic operators, handle memory-allocation related issues and take the best individual on the basis of the specified fitness function.

Figure 13: Evolve Diagram



3.3.2 Neural Network Module:

The NN module is the driving force for the autonomous agent and is responsible for the learning that takes place through out its lifetime. Following a specific research focus in this regard (being pursued by Joseba Urzelai, Nolfi et al), *XINC* employs a fully recurrent plastic neural network.

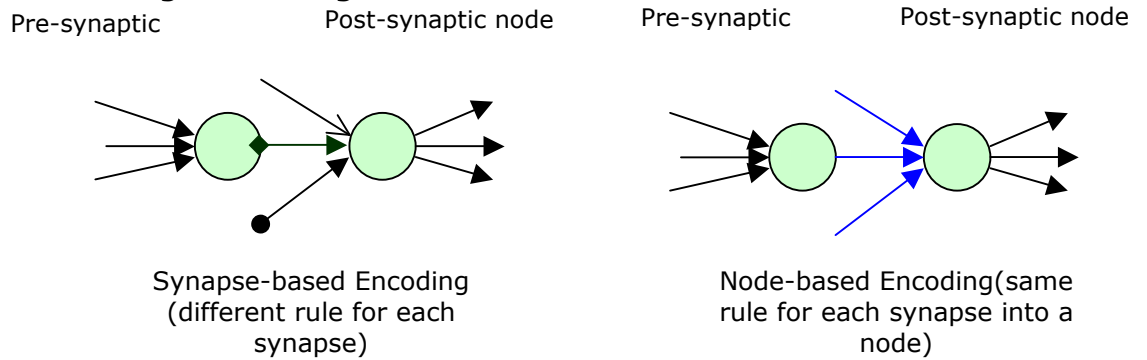
The network is designed such that it takes inputs in the form of a vector and operates on its own set of vectors to yield an output. The knowledge acquisition lifetime of a network spans the lifetime of the individual (this interaction with the evaluation module will be elaborated further). Various multidimensional arrays are used in the design to simulate the activations and recurrence of the network.

The design of the Neural Network Module is broken down into the following sub-modules:

3.3.2.1 Gene Decoding:

This sub-module handles the conversion of a genome into an operating neural network. Since the architecture of the neural network is fixed, the genome yields a set of learning rules that are applied onto the Neural Network nodes during its knowledge acquisition lifetime. The gene decoding component uses a synapse-based encoding scheme. A comparison of these schemes is as follows:

Figure 14: Weight Encoding



3.3.3 Evaluation Module:

The evaluation module integrates the GA and the NN modules. It is here that a robot is simulated in a test environment and its behavior is evaluated. This is basically a simulation that is object-oriented in its inherent design.

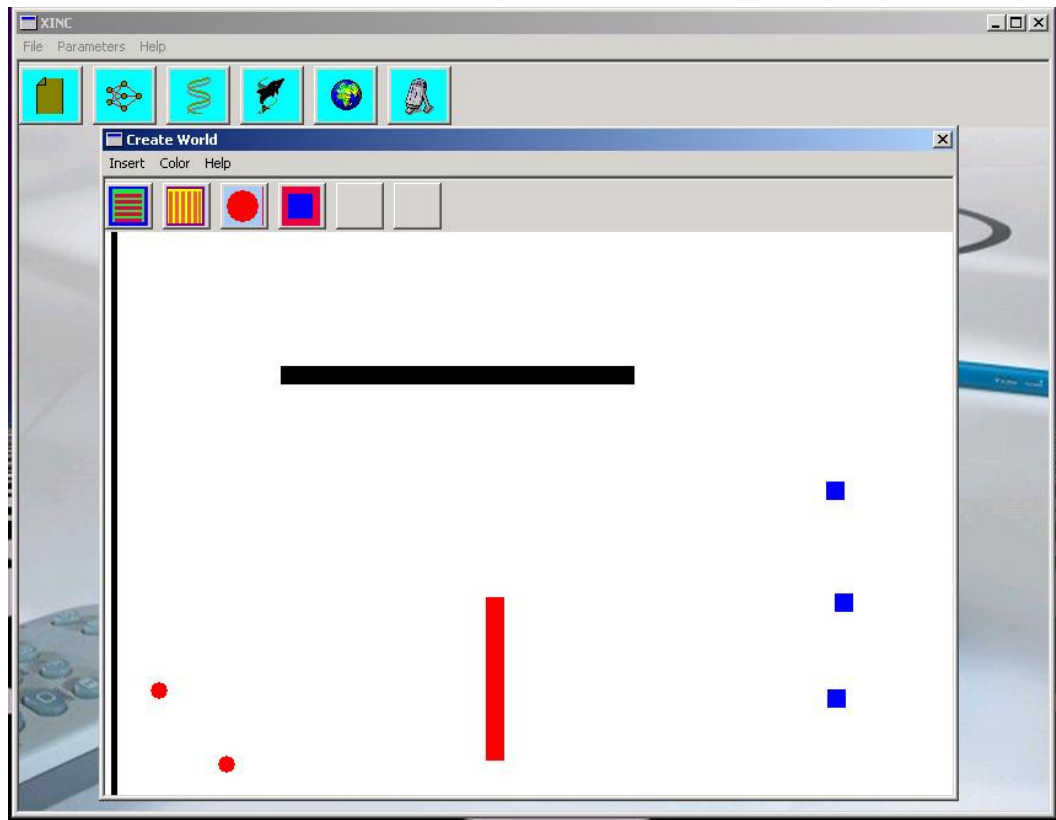
The method used in modeling the environment and the robots is the prototyping technique (Booch, Rumbaugh et al). The robot is modeled by the behavior of its sensors and its motors. The environment is mapped onto a Cartesian coordinate system with obstacles stored in independent vectors linked to the environment grid. The design of the simulation application is as follows:

3.3.3.1 World Creation:

Initially a world is created in which a simulated robot is allowed to interact and learn. The world comprises of a number of obstacles, placed arbitrarily. A number of different types of obstacles are provided, so that a generalized solution can be achieved.

The user is given a world creation facility in which he/she can create, edit and save any world that is required for any particular simulation. The world creation facility that has been designed and implemented in *XINC* is shown in the following screen shot.

Figure 15: Screen Shot #2 Create World



3.3.3.2 Robot Modelling:

The major portion of the simulation is the modeling of the robot. The physical characteristics of the robot such as the proximity sensors and the DC motors have to be modeled realistically. The first step of this robot modeling was the placement of the sensors and the wheels onto the robot. Once this has been achieved, the next phase is to maintain the positions of the sensors on the robot as the robot moves in the world. This requires a lot of manipulations, and trigonometric calculations.

3.3.3.3 Obstacle Detection:

Another major area of working is object detection. The robot should be aware of the obstacles as it moves in the world so that it can act accordingly. This depends heavily on how the obstacles are represented in the memory. The way we have done it is as follows. The obstacles are placed in the given area and their coordinates are added into a global obstacle array. As the robot moves this array is constantly monitored so that the obstacles can be detected. As a number of obstacles are supported, the mechanism of obstacle detection is different for each obstacle.

3.3.3.4 Remote Sensor Modelling:

The remote sensor modeling is another major component of the evaluation module. The sensors of the robot are vital for our applications performance. The real sensors have a physical field of detection and range. This range cannot be compromised in their simulation, so we cannot have a sensor with infinite range. In our simulation for a Hemisson robot, we set the range of each sensor's field of vision to be 30° with a attenuating sensitivity factor. The implication of this is that a sensor can detect only those obstacles that are within its field. A deviation from actual behaviour of the sensors in the simulation is that the sensors in the simulation behave linearly over distance, whereas actual sensors tend to have a different behaviour.

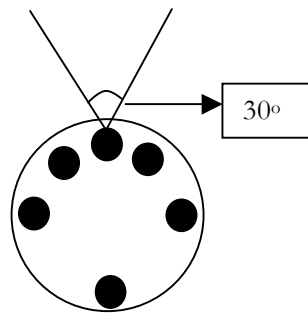


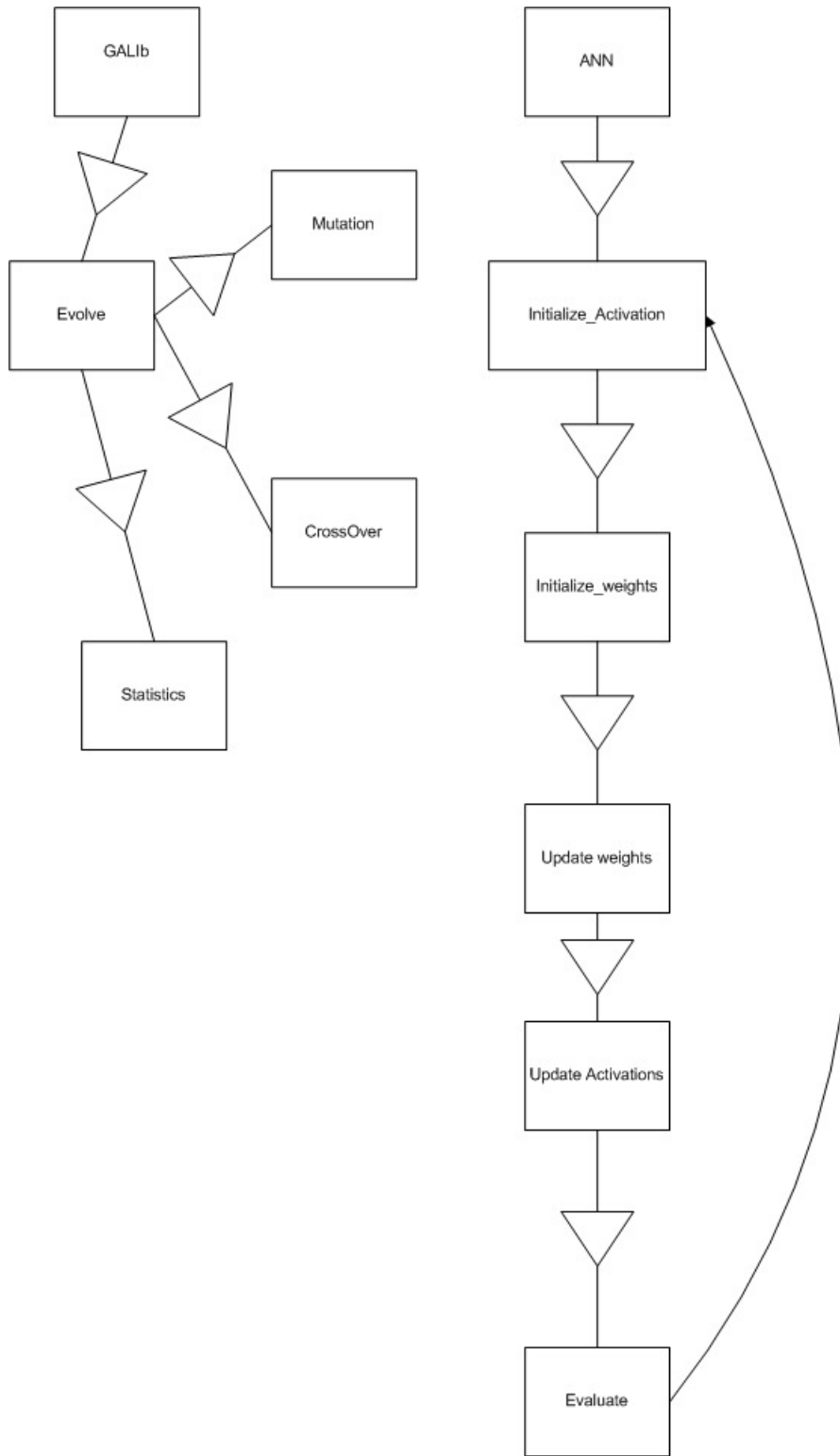
Figure 16: Viewing Angle

3.3.4 Deployment Module:

The deployment module is the final phase of the system. It transfers a trained and tested neural controller onto the real robot and is responsible for all the communication that takes place with the real robot. It translates a binary string genome into a neural network and copies it onto the microcontroller memory of a robot. This design is dependent on the interfacing features of the robot. Given the severe memory constraint imposed on our design by the memory size of the acquired robot, an automated deployment module was not realized, rather, the best controller was hard coded into the memory of the robot directly.

3.4 Information Flow:

With a design that incorporates numerous levels and spans across multiple domains, the entire flow of information can effectively be shown in a abstract manner of representation. The highest level representation of the flow of information is as follows:



**Figure 17:
Information
Flow**

3.5 Implementation Issues:

An overview of the major problems and their resolutions is discussed in this section. Relevant portions of the actual code discussed here can be found at in Appendix A.

3.5.1 Genetic Algorithm Implementation:

The Genetic Algorithm was not directly implemented and an open source library was used. This provided us with a less optimized and efficient solution but gave us time to focus on more important issues of the project. The library used was GALib, and it contained all major components of a Genetic Algorithm, coded in C++. It is freely available from: <http://lancet.mit.edu/ga/>. The main task that GALib performs is the creation of a representation and the corresponding genetic strings. The objective function was designed and coded separately and integrated with the GALib code segment.

3.5.2 Plastic Neural Network Implementation:

As plastic neural networks are inherently discrete-time recurrent neural networks in which each node is connected to every other node, the best way to model such a network was through matrices. By using a specific structure for each node, there would have been a lot of overhead in simulating the operation of such a network (n^2 operations per cycle). Since the architecture of the network that we used for evolution (Input: 6, Output: 2, Hidden <6) was small, the matrix form of implementation was quite efficient and suitable. The squasher function to scale all values was a binary sigmoid.

3.5.3 Rotation of Motors:

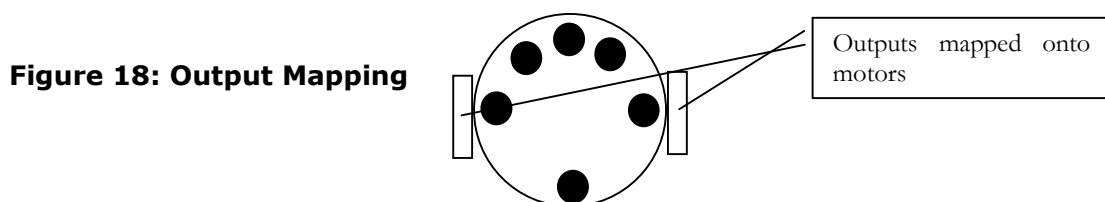
The rotation of the two wheels of the robot in simulation was done by mapping the output of the network to the two motors. In order to determine the next coordinate at which the robot would be after the specific number of rotations, we used the following formula:

$$\text{Angle of Robot's current direction} = \theta$$

$$\text{New Angle} = \theta + (\text{difference of outputs})$$

$$\text{New X-Coordinate} = \text{Previous X-Coordinate} + (\text{Sum of outputs}) + \cos(\theta/4)$$

$$\text{New Y-Coordinate} = \text{Previous Y-Coordinate} + (\text{Sum of outputs}) + \sin(\theta/4)$$

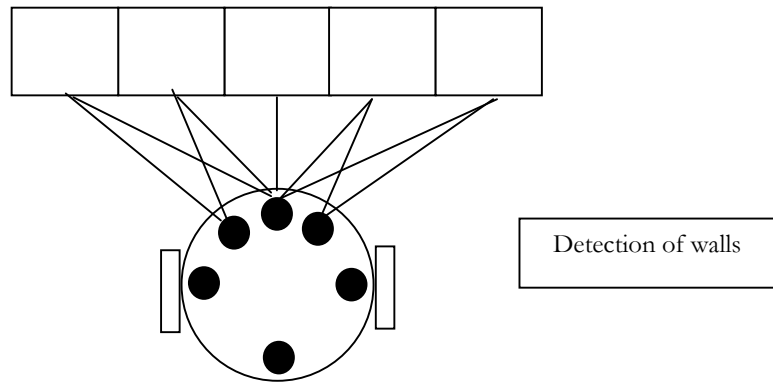


The corresponding code for this formula and its implementation is given in the appendix.

3.5.5 Obstacle Modelling:

The obstacles present in the environment were modeled by creating a global array which was checked at each step in the simulation. Obstacles were modeled as bins, squares and walls. In order to model a wall (which was actually a series of squares), the simulation would treat it as a set of squares and compute the distance and detection by the robot for each square separately. The drawback of this reliable technique was an increase in computations whenever a wall was encountered by the robot.

Figure 19: Wall Detection



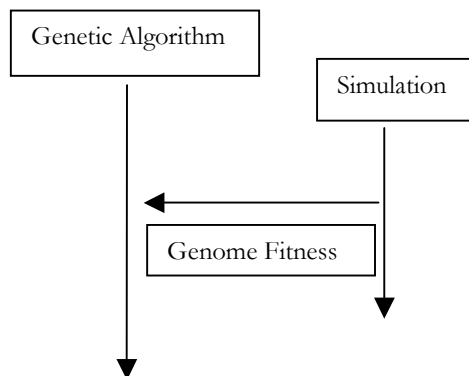
3.5.6 Random Number Generation:

Instead of employing the standard random number generating functions, we used a customized random number generator. The code for its implementation is given in the appendix.

3.5.7 Parallel Execution of GA and Simulation:

In order to have the GA concurrently run with the simulation, both of them were executed on different threads and then synchronized. This enabled the application to keep track of the identity of the genome currently being tested in simulation and then incorporate the results in the Genetic Algorithm running alongside.

Figure 20: Parallel Execution



3.5.8 Storage of Parameters of experiments:

In order to maintain a persistent set of parameters for the user each time he/she wants to conduct experiments, a certain file saving structure has been imposed in *XINC*. Each time a new project is created, the following files are created to store various parameters:

<project-name>.prj	-	Main Project File
<project-name>.xaf	-	ANN Parameters File
<project-name>.xef	-	Evolution Parameters File
<project-name>.xgf	-	GA Parameters File
<project-name>.xwf	-	World Parameters File

Chapter 4

Our Approach

This chapter describes our approach to the evolution of adaptive neural controllers for autonomous mobile robots. We suggest encoding mechanisms for parameter self-organization, instead of the parameters themselves as in conventional approaches. We argue that evolution of such adaptation mechanisms brings a number of advantages with respect to traditional evolution of neural controllers, and present a set of experiments in order to support our arguments.

In addition, we provide functional and behavioral analysis of evolved adaptive controllers aiming at understanding the role played by evolved self-organization mechanisms in the behavior displayed by the robot.

4.1 Evolution and Adaptation:

Artificial evolution of adaptive individuals can provide computational advantages and richer adaptive dynamics with respect to individuals whose defining parameters are entirely genetically-determined. The advantages amount to discovery of better solutions for a given problem, to faster convergence, and to improved robustness in face of changing fitness landscapes. They are thus relevant for artificial evolution of robotic control systems.

Most of the work done so far and effectively applied to robots, or realistically simulated organisms, shares two components: all synaptic weights are individually specified and directly encoded on the genetic string, and lifetime adaptation amounts to some standard gradient-descent learning algorithm. However, in the work by Floreano and Mondada (1996b, 1998), by Floreano and Nolfi (1997), and Joseba Urzelai (2000), instead of simply combining off-the-shelf evolutionary and learning algorithms, a different approach where synaptic weights were not genetically specified and lifetime adaptation consisted of Hebbian synaptic changes was proposed. In this approach it has been mentioned that evolved adaptive individuals display more robust behaviors than traditional genetically-determined controllers (Floreano & Mondada, 1996b) and win tournaments in a competitive co-evolutionary scenario (Floreano & Nolfi, 1997).

In this chapter we describe the approach in detail and introduce a compact genetic representation that encodes for adaptive properties of each neuron of the control architecture, instead of adaptive properties of individual synapses of the neural network.

4.1.1 Encoding Mechanisms of Adaptation:

The artificial chromosomes encodes a set of four modification rules for each component of the neural network (components can be individual synapses or groups of synapses that converge towards the same

neuron), but not the synaptic strengths of the network. Whenever an artificial chromosome is decoded into a neural controller, the synaptic strengths are set to small random values. This means that the robot will initially display random actions both at generation 0 and at later generations. However, as time goes the synapses start to change their value using the genetically specified rules every 100 ms (the time necessary for a full sensory-motor loop on the physical robot). Notice that synaptic adaptation occurs on-line while the robot moves and that the network self-organizes without external supervision and reinforcement signals. The fitness function is evaluated along the whole duration of each individual's "life". This introduces an implicit learning cost (Mayley, 1996) that gives selective advantage to individuals that can adapt faster. At the end of the life, the final synaptic strengths are not "written back" into the artificial chromosome.

We have selected four types of modification rules to be encoded on the artificial chromosome. The choice has been based on neurophysiological findings and on computational constraints of local adaptation. In other words, these rules capture some of the most common mechanisms of local synaptic adaptation found in the nervous systems of mammals (Willshaw & Dayan; 1990). These rules have been modified in order to satisfy the following constraints, synaptic strength cannot grow indefinitely, but is kept in the range $[0, 1]$ by means of a self-limiting mechanism which depended on synaptic strength. Because of this self-limiting factor, a synapse cannot change sign, which is genetically specified, but only strength. Each synaptic weight w_{ij} is randomly initialized at the beginning of the individual's life and can be updated after every sensory-motor cycle (100 ms),

$$w_{ij}^t = w_{ij}^{t-1} + \eta \Delta w_{ij},$$

where $0.0 < \eta < 1.0$ is the learning rate and Δw_{ij} is one of the four modification rules specified in the genotype:

1. Plain Hebb rule: can only strengthen the synapse proportionally to the correlated activity of the pre- and post-synaptic neurons,

$$\Delta w = (1 - w) xy$$

2. Postsynaptic rule: behaves as the plain Hebb rule, but in addition it weakens the synapse when the postsynaptic node is active but the presynaptic is not.

$$\Delta w = w(-1 + x)y + (1 - w)xy$$

3. Presynaptic rule: weakening occurs when the presynaptic unit is active but the postsynaptic is not.

$$\Delta w = wx(-1+y) + (1 - w)xy$$

4. Covariance rule: strengthens the synapse whenever the difference between the activations of the two neurons is less than half their maximum activity, otherwise the synapse is weakened. In other words, this rule makes the synapse stronger when the two neurons have synchronous activity.

$$\Delta w = (1 - w) F(x, y) \text{ if } F(x, y) > 0$$

$$\Delta w = (w) F(x, y) \quad \text{otherwise}$$

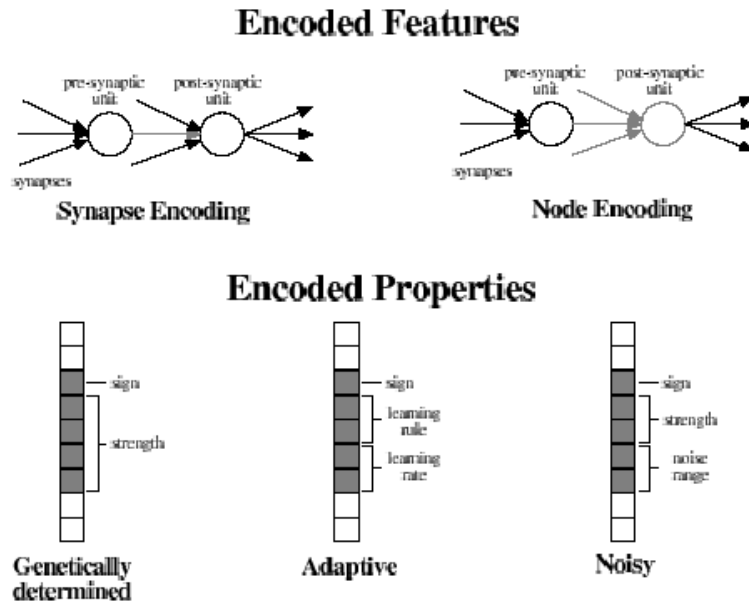
where $F(x, y) = \tanh(4|x-y|)-2$ is a measure of the difference between the presynaptic and postsynaptic activity, $F(x, y) > 0$ if the difference is bigger or equal to 0.5 (half the maximum node activation) and $F(x, y) < 0$ if the difference is smaller than 0.5.

The genetic encoding refers to the way in which the neural controller is mapped into a bit string that represents the artificial chromosome of an individual. A gene is a set of bits that encode a given feature of the neural controller. We consider two aspects of genetic encoding: the feature level and the properties of that feature.

Features:

We consider two levels of feature encoding that reflect the two basic components of a neural network: the synapses and the nodes. Synapse Encoding refers to the case where a gene encodes the properties of an individual synapse. In this case, the artificial chromosome will have as many genes as synapses in the network. This type of encoding is rather common in works that combine evolutionary computation and neural networks; it is also known as "direct encoding" (Yao, 1993). Node Encoding instead refers to the case where a gene encodes the properties of an individual node. In that case, all the incoming synapses to that node will share the same properties specified for that node (except for the sign of the traversing signal which is a property of the presynaptic node). The artificial chromosome will have as many genes as nodes in the network. Synapse Encoding allows a detailed definition of the neural network, but for a fully connected network of N neurons the genetic length is proportional to N^2 . Instead Node Encoding requires a much shorter genetic length (proportional to N), but it allows only a rough definition of the controller.

Figure 21: Encoded Features



Properties:

Irrespectively of the feature level chosen (synapses or nodes), each gene is composed of five bits that represent the properties of the corresponding feature. We consider three types of properties (see bottom of figure above and table below). For all three types, the first bit always represents the sign of the signal traveling outward (either through the synapse in the case of synapse encoding or through the outgoing axon in the case of node encoding). The remaining four bits can encode the following properties:

1. Genetically-determined synapses:
4 bits encode the synaptic strength. This value is constant during "life" of the individual. This is the conventional way of evolving neural networks (Yao, 1993).

Figure 22: Bit Encoding

Encoding	Bits for one synapse / node				
Genotype	1	2	3	4	5
A	sign	strength			
B	sign	Hebb rule		rate	
C	sign	strength		noise	

2. Adaptive synapses:
2 bits encode the 4 adaptive rules described above and 2 bits the corresponding learning rate. Synaptic weights are always randomly initialized at the beginning of an individual's life and then updated on-line

every 100 ms according to their own adaptation rule while the individual interacts with the environment. This is the core of the methodology proposed in this chapter, which is evolving the mechanisms of self-organization of a controller.

3. Noisy synapses:

2 bits encode the weight strength and 2 bits a noise range. The synaptic strength is genetically determined at birth, but a random value extracted from the noise range is freshly computed and added every 100 ms while the individual interacts with the environment. This is a control condition to check whether the effects of random variations are equal or different from the mechanisms of self-organization described above.

In previous work, Floreano and Mondada (1998) used only Synapse Encoding and showed that evolution of adaptive synapses produces in less generation better controllers than evolution of genetically-determined synapses for simple reactive navigation. After that Joseba Urzelai worked on using Node Encoding with the adaptive synapses which is a much more compact representation. We have followed that approach in our project.

4.2 Task: Obstacle Avoidance:-

In our project we have compared the performance of evolutionary adaptive controllers with respect to evolution of synaptic weights and evolution of noisy synapses in a sequential task.

A mobile robot Hemisson equipped with IR sensors is positioned in the rectangular environment. Many obstacles are placed in the environment, which are of different shapes. Their color is chosen to be white as the IR sensors of the robot detect the white color most convincingly. Each individual of the population is tested on the same robot, one at a time, for 500 sensory motor cycles, each cycle lasting 100 ms. At the beginning of an individual's life, the robot is positioned at a random position and orientation.

The fitness function is given by the number of sensory motor cycles spent by the robot in the environment avoiding the obstacles divided by the total number of cycles available (500). In order to maximize this fitness function, the robot should avoid the obstacle and move in the environment freely. Since this sequence of actions takes time (several sensory motor cycles), the fitness of a robot will never be 1.0. Also, a robot that cannot manage to complete the entire sequence will be scored with 0.0 fitness. The six infra-red sensors detect the obstacle and fitness value is calculated based on those values. The output of the sensors is given as input to the neural controller. After 500 sensory motor cycles; the robot is stopped and then again started applying random speeds to the wheels for 5 seconds.

Notice that the fitness function does not explicitly reward this sequence of actions (which is based on our external perspective), but only the final outcome of the sequence of behaviors chosen by the robot. This

function is behavioral, internal, the computation is based on variables read through the sensors of the robot.

The controller is a fully-recurrent discrete-time neural network. It has access to one type of sensory information from the robot.

1. Infra-red light:

The active infrared sensors positioned around the robot measure the distance from objects. Their values are pooled into directly given to the inputs of the neural controller.

Two motor neurons are used to set the rotation speed of the wheels, by mapping the continuous activation of each neuron, normalized between 0 and 1, to a discrete speed value between -20 and 20 (negative values for backward rotation, and positive values for forward rotation). Neurons are updated every 100 ms according to the following equation

$$y_i \leftarrow \sigma \left(\sum_{j=0}^N w_{ij} y_j \right) + I_i,$$

where y_i is the activation of the i^{th} neuron, w_{ij} is the strength of the synapse between presynaptic neuron j and postsynaptic neuron i , N is the number of neurons in the network, $0 < I_i < 1$ is the corresponding external sensory input, and $\sigma(x) = (1 + e^x)^{-1}$ is the sigmoidal activation Function. $I_i = 0$ for the motor neurons.

Each synaptic weight w_{ij} is randomly initialized at the beginning of the individual's life and can be updated after every sensory-motor cycle (100 ms),

$$w_{ij}^t = w_{ij}^{t-1} + \eta \Delta w_{ij}$$

where $0.0 < \eta < 1.0$ is the learning rate and Δw_{ij} is one of four modification rules specified in the genotype.

4.3 Analysis of Results:

We have found through the results of the experiments that evolution of adaptive synapses brings a number of advantages with respect to traditional evolution of synaptic weights. It can generate viable controllers in much less generations and the evolved controllers display more performant behaviors. Since adaptive synapses need not be specified on the genetic string because their strength is always randomly initialized at the beginning of an individual's test, this approach can rely on a very compact genetic encoding that specifies only the adaptive properties of individual nodes. Such a compact encoding scales up very well to

large networks with many synapses and allows evolution to find performant solutions using relatively small genetic populations.

The data obtained from control experiments with noisy synapses and from behavioral tests of evolved individuals with adaptation disabled, together with the behavioral analysis related to synaptic activity of adaptive individuals, all suggest that Hebbian adaptation plays a specific role in the functioning of the controllers both during evolution and during the life of an individual. In addition, we have showed that evolved individuals present specific adaptation rule patterns in their motor neurons, which are directly related to the behavior displayed by the robot.

When describing our controllers with changing synapses, we have never used the term learning to describe the behavior of individuals with changing synapses. The reason is that we have no evidence that evolved individuals are effectively learning something in the cognitive sense of acquiring new competences and storing knowledge. Instead, we used the term "self-organization" and "adaptation" to indicate that the modifications induced by the genetically evolved combination of rules produce behavioral changes that are functionally related to the fitness criterion.

The approach to the evolution of adaptive systems presented in this chapter differs in several aspects from the behavior-based approach:

- (i) Here, the human designer provides only a general evaluation function that evolution exploits to generate control architecture, whereas in the behavior-based approach the human designer plays a main role in the construction of the control architecture;
- (ii) In the approach described in this chapter, lifetime adaptation is shaped by evolution and does not require any supervision signals. Instead, in the behavior-based approach the human designer must decide which components of the control architecture may need learning and provide them with a supervision program designed by hand;
- (iii) In the behavior-based approach individuals are selected for their ability to behave efficiently, whereas here individuals are selected for their ability to adapt their behavior to the environment (synaptic strengths are initialized to random values at the beginning of individual's life).

We believe that synaptic plasticity generated by the evolution of adaptation rules might increase the robustness of robot controllers to unpredictable sources of change (Floreano & Nolfi, 1997 and Joseba 2000). In the next section we try to verify the adaptation capabilities of our approach to environmental changes.

4.4 Evolution in changing environments

The situated nature of Evolutionary Robotics is such that often evolved controllers find surprisingly simple yet efficient solutions that capitalize upon unexpected invariants of the interaction between the robot and its environment.

For example, a robot evolved for the ability to discriminate between shapes can do so without resorting to expensive image processing techniques by simply checking the correlated activity of two receptors located in strategic positions on the retinal surface (Harvey et al., 1994).

Analogously, a robot evolved for finding a hidden location can display the performances similar to those obtained by rats trained under the same conditions without resorting to complex environmental representations by using simple sensory-motor sequences that exploit geometric invariants of the environment (Lund & Miglino, 1998).

The remarkable simplicity and efficiency of these solutions is a clear advantage for fast and real-time operation required from autonomous robots, but it raises the issue of robustness when environmental conditions change. Environmental changes can be a problem also for other approaches (programming, learning, e.g.) to the extent in which the sources of change have not been considered during system design; but they are even more so for evolved systems because these often rely on environmental aspects that are often not predictable by an external environmental observer.

Environmental changes can be induced by several factors such as modifications of the sensory appearance of objects (e.g., different light conditions), changes in sensor response, re-arrangement of environment layout, transfer from simulated to physical robots, and transfer across different robotic platforms.

In the previous section about evolution and adaptation, we have suggested to evolve the adaptive characteristics of a controller instead of combining evolution with off-the-shelf learning algorithms. The method consists of encoding on the genotype a set of four local Hebb rules for each synapse, but not the synoptic weights, and let these synapses use these rules to adapt their weights online starting always from random values at the beginning of the life.

Since the synaptic weights are not encoded on the genetic string, there cannot be genetic assimilation of abilities developed during life. In other words, these controllers can rely less on genetically-inherited invariants and must develop on-the-fly the connection weights necessary to achieve the task.

At the same time, the evolutionary cost of adaptation (i.e., the time and energy spent to adapt goes to the detriment of the individual's fitness) implicitly puts pressure for the generation of fast adaptive architectures.

The results reported in the previous section have shown that the evolution of adaptive controllers generates better performances in less generations than evolution of genetically- determined weights on a sequential navigation task.

Here we describe another experiment designed to test robustness of this approach to environmental changes that were not included during evolutionary training. We focus on two major sources of environmental change: on-line sensory-motor adaptation (namely new sensory appearances, transfer from simulations to real robots,) and adaptation to new spatial relationships.

We have made all these environment changes for Hemisson and the results were very encouraging. It was able to adapt to the changing environments and even to dynamically changing environments. We then deployed that neural controller on a real robot and its behavior was very satisfactory. The real robot was also able to adapt to the new spatial relationships.

Chapter 5

Conclusion

In the experiments carried out in the "Obstacle-Avoidance" environment we have studied adaptation to new sensory appearances by testing evolved individuals in environments in which color of the walls was different from that used during evolution, and by transferring individuals evolved in simulation to a physical robot. The results show that genetic encoding of adaptation mechanisms supports evolution of controllers with better adaptation capabilities than genetic encoding of the parameters themselves. Adaptive individuals are capable of successfully performing in environments that are different from that used during evolution by adapting their strategy to the new constraints of the environment. Instead, genetically-determined individuals often fail in adapting to different environments because their behavior is tightly coupled to the characteristics of the environment used during evolution.

We think that the approach presented here represents a significant step forward towards making Evolutionary Robotics applicable to real-world applications of autonomous robotics. In scenarios like those for example of robots probing an asteroid surface or robots interacting with an handicapped person it is impossible to evolve the control system on the spot (not even incrementally). However, one might reproduce the working conditions in the laboratory to some degree of approximation and evolve the adaptive controller in there. The controller would then be transferred on the final robot and let free to adapt to actual working conditions in a few seconds.

5.1 Contributions:

We believe that the main contributions of this thesis are:

1. Different behavioral, learning, and evolutionary approaches to the creation of autonomous robots have been described and compared, and a need for self-organizing mechanisms capable of coping with, unpredictable sources of change has been identified.
2. The advantages and limitations of combining artificial evolution and lifetime learning have been reviewed in the context of autonomous robots working in dynamic environments.
3. A new compact genetic specification of neural controllers suitable for evolution has been conceived.
4. Evolution of local adaptation rules for robot neural controllers has been investigated, and has proven to be more powerful than traditional evolution of synaptic weights in what concerns performance of evolved solutions, number of generations needed to evolve the solutions, and scalability to larger networks.

5. Several experiments have been performed aiming at investigating the adaptive capabilities of evolved robot controllers. Several sources of change, such as new sensory appearances, environmental rearrangements, and transfer from simulation to a physical robot have been considered in order to allow the advantages of the evolution of adaptation rules with respect to the evolution of synaptic weights.

5.2 Limitations:

Although compact encoding of local adaptation rules has proven to be very efficient in coping with environments that differ in significant ways from that used during evolution, it is not at all evident how to analyze evolved adaptive controllers. Earlier we reported some results that indicate that modifications induced by the genetically evolved combination of rules produce behavioral changes that are functionally related to the fitness criterion. However, further work is needed in order to establish how specific combinations of rules affect the behavior of the robot. It would be interesting to apply this approach to other *tasks*, in order to study whether specific kind of rules are needed to generate a given behavior.

All the results reported in this report compare evolution of local adaptation rules with evolution of synaptic weights. It has been clearly shown that the former outperforms the latter in every aspect considered in this work. However, it would be interesting to compare evolution of adaptation rules with other approaches combining evolution and learning, specially in what concerns adaptation to changing environments. When describing our controllers with changing synapses we have never used the term *learning* to describe the behavior of individuals with changing synapses. The reason is that we have no evidence that evolved individuals are effectively learning something in the cognitive sense of acquiring new competences and storing knowledge. Instead, we induced by the genetically evolved combination of rules produce behavioral changes that are functionally related to the fitness criterion.

Some preliminary experiments (not represented here) has some difficulties to solve tasks in which long-term memory is required. In fact, this approach is capable of generating fast changes in the control architecture, but some other mechanism seems to be necessary to fix these changes in order to implement long-term memory.

5.3 Future Directions:

There are many interesting issues that could be addressed to further investigate into the evolution of adaptation mechanisms for robot neural controllers. A possible further research direction, is concerned with the long-term memory issue mentioned above, the adaptive neural controller proposed here performs fast synaptic adaptation in order to imitations that may be described as learning. To this end, some architectural variations such as lateral modulatory connections (Kay, Floreano, & Phillips, 1997) or synapse-to-synapse inhibitory connections (Kording, K and Konig, P., 2000) may be considered. As mentioned earlier, we think that our adaptive strategy can be useful for evolving more complex and powerful neural morphologies. In current methods there is a trade-off between the complexity of the genotype to phenotype mapping and the evolvability of the system, partly due to the fact that the phenotype largely depends on genetic instructions. By evolving the adaptive characteristics along with other high-level parameters (e.g., position and type of nodes) of the controller and by letting the final structure develop in close interaction with the environment, simpler genetic encodings and higher tolerance to mutations can be obtained, This would make the evolved controllers more viable add neutrality to the genetic landscape, and ultimately improve evolvability.

Appendices

A.1 Code for generating random weights for initial startup of simulation:

```
#include <limits.h> // defines INT_MAX and ULONG_MAX constant
#include <time.h> // defines time() function

class Random
{ public:
    Random(long seed=0) { _seed = ( seed?seed:time(NULL) ); }
    void seed(long seed=0) { _seed = ( seed?seed:time(NULL) ); }
    int integer() { return _next(); }
    int integer(int min, int max)
        { return min + _next()%(max-min+1); }
    double real()
        { return double(_next())/double(INT_MAX); }
private:
    unsigned long _seed;
    void _randomize()
        { _seed = (314159265*_seed + 13579)%ULONG_MAX; }
    int _next()
        { int iterations = _seed % 3;
          for (int i=0; i <= iterations; i++) _randomize();
          return int(_seed/2); //Return Random number
        }
};
```

A.2 Code for calculating new position of Robot after output has been mapped onto motors:

```
void RotateMotors(HWND hWnd, double motor1, double motor2)
{
    //double x, y;
    RX = robot->x; //Get current X,Y coordinates of robot
    RY = robot->y;
```

```
double Delta_direction;
//The following conditional statements are included to check for outputs that are out of range or
//erroneous, and put them into scale
if(motor1>9)
    motor1 = 9;
if(motor1<-9)
    motor1 = -9;
if(motor2>9)
    motor2 = 9;
if(motor2<-9)
    motor2 = -9;

if(motor1<=0 && motor2<=0)
    motor1 = 1;

Delta_direction = motor2 - motor1;
Delta_direction = Delta_direction / 200;

theta = theta + Delta_direction;
theta = NormRad(theta);
double a1, a2;
a1 = RX;
a2 = RY;

RX = RX + (motor1 + motor2) * cos( theta ) / 4;
RY = RY - (motor1 + motor2) * sin( theta ) / 4;
theta = theta + Delta_direction;
theta = NormRad( theta );
{
    ClearScreen(hWnd);
    //Sleep(10);
    MoveRobot(hWnd,RX, RY);
}
}
```

Bibliography

1. Evolutionary Robotics: Coping with Environmental Change. *Urzelai, J. and Floreano, D.* ECAL 2000.
2. Reinforcement Controller Design in a Simulation Environment. *Spronck, P.H.M. and Kerckhoffs E.J.H.* 1997.
3. Ago Ergo Sum. *Floreano, D.* 1999.
4. Evolutionary Learning of a neural robot controller. *Spronck, P. Ida G. Sprinkhuizen-Kuyper, Eric O. Postma.* 1997.
5. Evolving and breeding robots. *Lund, H. Miglino, O.* 1998.
6. Evolving variable plasticity in Neural Systems. *Bullinaria, J.* 1997.
7. Evolution of neural control structures: Some experiments on mobile robots. *Floreano, D. Mondada, F.* 1996.
8. Robot Shaping: Developing autonomous agents through learning. *Dorigo, M. Colombetti, M.* 1994
9. Evolving Plastic neural network controllers for autonomous robots. *Flotzinger, D.* 1996
10. Evolving a learning algorithm for the binary perceptron. *Fontanari, J. F. Mei, R.* 1991
11. Relearning and evolution in neural networks. *Harvey, I.* 1996.
12. Evolutionary Robotics: The Sussex Approach. *Harvey, I. Husbands, P. Cliff, D. Thompson, A. Jakobi, N.* 1997.
13. Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot. *Floreano, D. Mondada, F.* 1994.
14. Bio-Inspired Computing Systems: Towards Novel Computational Architectures. *Mange, D. Tomassini, M.* 1997.
15. Optimal unsupervised learning in a single-layer feedforward neural network. *Sanger, T.D.* 1989.
16. Incremental Evolution with Minimal Resources. *Urzelai, J. Floreano, D.* 1999.
17. Challenges in Evolving Controllers for Physical Robots. *Mataric, M. Cliff, D.* 1996.
18. Developing Neural Network Controllers for Robots. *Meeden, L.* 1996.
19. Evolving non-trivial behaviour on autonomous robots: Adaptation is more powerful than decomposition and integration. *Nolfi, S.* 1997.
20. Learning and Evolution in Neural Networks. *Nolfi, S. Elman, J. Parisi, D.* 1994
21. A comparison of matrix rewriting versus direct encoding for evolving neural networks. *Siddiqi, A.A. Lucas, S.M.* 1998.
22. Introduction to Reinforcement Learning. *Sutton, R. Barto, A.* 1998.
23. A review of evolutionary artificial neural networks. *Yao, X.* 1993.
24. Evolutionary Robots with On-line Self-Organization and Behavioral Fitness. *Floreano, D. Urzelai, J.* 1999.
25. Toward Indoor Flying Robots. *Nicoud, J.D. Zufferey, J.C.* 2002.
26. Hybrid, Metric-Topological, Mobile Robot Navigation. *Tomatis, N.* 2000.