Neubot Project: Framework for Simulating Modular Robots and Self-Organisation of Locomotion under Water

---

Barthélémy von Haller

February 18, 2005

BIRG, Logic Systems Laboratory (LSL)
School of Computer and Communication Sciences
Swiss Federal Institute of Technology Lausanne

Supervision:
Prof. Auke Jan Ijspeert

Abstract

The Biologically Inspired Robotics Group (BIRG) of the Swiss Federal Institute of Technology in Lausanne (EPFL) in collaboration with the Automated System Laboratory (ASL) of the EPFL is currently developing new hardware and software for an experimental modular robotic platform. Modular robots are robots built of several building blocks called *modules*. Usually modules are all similar, but this assertion is not yet true in this new project as the name implies: Neubot i.e. Von Neumann Robot. Like the von Neumann automata was made up of 28 different modules, the Neubot Project plan to develop several types of modules, each with its own abilities and capacities. When several modules are put together, efficient locomotion can be accomplished. This study describes the framework developed to simplify the physics simulations of modular robots, and in a second part this study will explore the self-organization of underwater locomotion. We will use the *Central Pattern Generator* (CPG) concept - inspired from the nervous system of vertebrates - based on oscillators made of neural networks in order to achieve locomotion. Moreover, we will use *co-evolution* of body and brain to create not only the controller but also new shapes adapted to the Neubot module.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Objectives

This project is formed of two part. The first one is to develop a complete, efficient and reusable framework for the simulation of modular robots. It must be easy to use and to achieve new types of modules. It was not developped only for the Neubot Project but with the intention to use it with any new type or concept of modules. The aim of the second part of the project is to achieve underwater locomotion with the Neubot modules assembled as a robot. I used co-evolution to create both the configuration of the robot and his controller. The controller is based on the CPG's concept and neural networks will be used as oscillators.

1.2 Modular Robots

Modular robots are robots built of multiple building blocks, identical in most cases, called *modules*. In the real world, we find a lot of examples of animals (essentially socials insects) like bees, termites or ants that can achieve feats that would be impossible by a single individual. The group work of the individuals offers new and very efficient possibilities for the whole colony. In modular robotics it's the same idea that we try to use. Many simple adaptive units interacting with each other are claimed to have potential advantages in robustness over traditional robotics, mainly in terms of adaptability, versatility and self-repair.

More precisely, modular robots can perform *self-reconfiguration* by changing their shape and adapt to different kinds of terrain and situations. For example, a modular robot can transform into a quadruped to cross over an obstacle then later transform into a ball to go more quickly to another location on a declivity. Another important property of modular robots is *self-reparation*. Like their biological counterparts, the modular robots are built upon redundant and decentralized architecture. Therefore, it is possible to excise a damaged module and replace it with another one if available.

1.3 CPG's and Oscillators

In 1966, experiments showed [24] that, in a decerebrated cat, a simple electrical stimulation of the brainstem was able to induce walking. Furthermore, an increase of the signal strength changed the walk velocity until the transition from walk to trot happened automously. These experiments demonstrated that the brain is not involved in the generation of the rhytmic signals that produce locomotion in the cat.

Later, Grillner explained [25] that the different gaits (walk, trot and gallop) are generated by *Central Pattern Generator* (CPG) located in the spinal cord. The CPGs are *Neural Networks* that generate oscillatory signals from a tonic input coming from the brain. These signals command the muscles. Thus the brain appears to play a high-level role in the locomotion, giving instruction regulating the beginning, velocity and termination of the locomotory activity. We will use this concept in order to achieve locomotion of our modular robots.

1.4 Co-Evolution

Traditionally, the biologically inspired learning technics and evolutionary algorithms only affects the controller of the robot. The hardware - shape of the robot, mode of propulsion, size and/or sensors - are designed by humans. This difference is probabily due to practical reasons: building and testing new evolved hardware is more complicated than doing the same for a controller on a preexistent robot.

Nevertheless, in nature, the controller (the neural system) and the structure (the body) are evolved in a parallel way. A little change of one leads to an adaptation of the second and this leads to a coherent and adapted to its environment life system. We never see an animal with suddenly a new limb and then slowly develop the required sensors and controller for it! Thus, the co-evolution of body and brain (other co-evolutions can be used as sensors-brain or predator-prey) seems to be a good way to find well adapted robot for a certain environment or a certain task.

1.5 Overview

The next chapter gives an overview of existing project that deal with modular robotics or co-evolution. It will expose their properties, advantages and interesting ideas as well as the main problems encountered. An in-depth description of the Neubot Project is then presented in chapter 3 (page 17).

Chapter 2

State-of-the-Art

2.1 Karl Sims block creatures

Already in 1994, Karl Sims co-evolved body and brain of block creatures (Figure 2.1). The morphology of these creatures and the neural systems for controlling their muscle forces are both genetically determined, and the morphology and behavior can adapt to each other as they evolve simultaneously.
The genotypes are structured as directed graphs of nodes and connections, and they can efficiently but flexibly describe instructions for the development of creatures' bodies and control systems with repeating or recursive components. When simulated evolutions are performed with populations of competing creatures, interesting and diverse strategies and counter-strategies emerge [6].

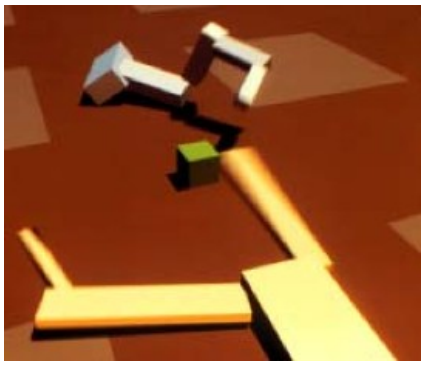

Figure 2.1 : *The Karl Sims block creatures competing over control of the box in the middle.*

2.2 Others projects concerning co-evolution

Numerous research groups use and study the co-evolution of body and brain since the work of K. Sims. Amongst all, Framesticks, a three-dimensional life

simulation project, offers various genotypes and fitness functions to co-evolve morphology and control of virtual stick creatures [7] [8]. In addition, Hornby and Pollack undertook many research so much on the encoding [9] (use of a generative encoding rather than of a direct encoding) that on the co-evolution in general [10]. Finally, other interesting projects can be found in [11][12].

2.3 M-TRAN

One of the most accomplished modular robotic projects is the M-TRAN (Figure 2.2) developed by the Distributed Systems Design Research Group of the National Institute of Advanced Industrial Science and Technology of Japan. M-TRAN is a modular robot able to achieve self-reconfiguration without human intervention.

The modules consist of two semi-cylindrical parts that are connected by a hinge. They can connect to other modules by magnetic force and disconnect from other by heating a Shape Memory Alloy coil. Each M-TRAN module has two motors which provide it with two degrees of freedom in the same plane [16] [17] [18].

Figure 2.2 : *Transformation of the Modular Transformer (M-TRAN) from crawler to a four-legged walking robot.*

2.4 PolyBot

The XEROX Palo Alto Research Center (PARC) developed a modular robot named PolyBot (Figure 2.3). The modules of the first generation were manually

screwed together, so they were not able to self-reconfigure . The PolyBot modules of third generation are able to self-reconfigure and dock, and have joint angle sensors, accelerometers and infrared proximity sensors used principally to aid in docking two modules. However, each module has only two connection surfaces and therefore mostly snake-like configurations are possible unless passive elements are used (connection modules). The goal of this third generation is to demonstrate locomotion with large configurations and reconfiguration between configurations. PolyBot has shown versatility and adaptability, by demonstrating locomotion over a variety of terrain [19].

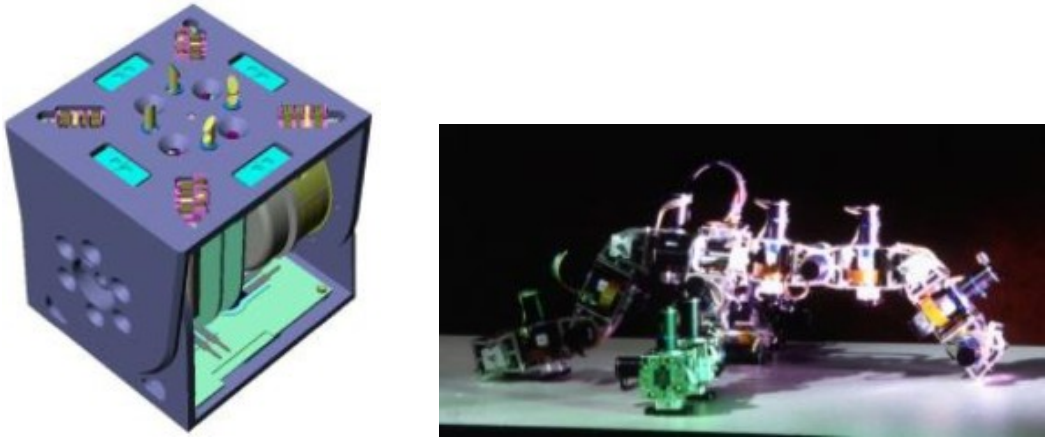

Figure 2.3 : *Left: PolyBot Generation 3 module. Right: PolyBot G2 with 24 modules in a four-legged spider configuration.*

2.5 CONRO

The CONRO Project from the Information Science Institute of the University of Southern California has a goal of providing the Warfighter with a miniature reconfigurable robot that can be tasked to perform reconnaissance and search and identification tasks in urban, seashore and other field environments [22]. The base topology is simply connected, as in a snake, but the system can reconfigure itself in order to grow a set of legs or other specialized appendages (Figure 2.4). Each module will consist of a CPU, some memory, a battery, and a micro-motor plus a variety of other sensors and functionality, including vision and wireless connection and docking sensors.

Unlike the other projects, CONRO modules do not have hermaphroditic connectors but rather one female and three male connectors. CONRO robots are able to discover autonomously the way they are connected through an hormone inspired decentralized communication system. Much like natural hormones, artificial hormones allow different types of responses from differents parts of the robot's body.[20] [21].

Figure 2.4 : *CONRO self-reconfiguring itself from a snake to a two armed crawler configuration.*

2.6 Yamor

BIRG recently started developing prototypes for modular robot units called YaMoR (for Yet another Modular Robot) (Figure 2.5). The aim of the project is to create robot units that can rapidly be attached to each other in order to create arbitrary multi-unit robot structures.

The units have the following characteristics: (1) each unit is autonomous in terms of power, sensing, actuation, and computing, (2) they are driven by heavy-duty servos such that one unit can lift up to 3 others, (3) they communicate via BlueTooth (i.e. no need for electrical connections between units), and (4) they are equiped with an FPGA for providing flexible computational power.
The unit are designed to be of general-purpose [27].

This project is yet well advanced but less complex than the Neubot project. As the modules are unable to connect or disconnect to or from each other, the self-reconfiguration and self-assembling are impossible.

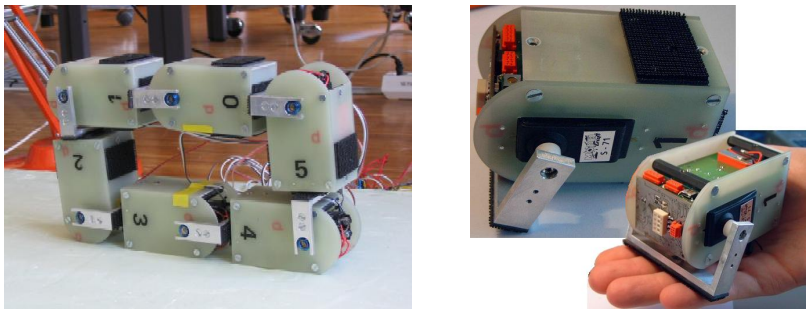

Figure 2.5 : *YaMoR modules. Left: assembled. Right: alone.*

2.7 Hydra, HYDRON, Swarm-bots

The partners of the HYDRA project [23] want to create a fundamentally new design standard, which will allow end-users to design new artefacts in an easy manner by using an open standard architecture made up of simple building blocks,

simple connections, and simple interactions. This means that systems comprising hundreds of basic building blocks can become self-assembling, self-repairing and differentiated on location.

Inspired by developmental biology, the HYDRA partners want to develop a physical model of self-assembling cells in 3D space. Each cell will cooperate with other cells and will be able to make choices as to investment in movement, perception and communication. It is expected that this design will lead to task specialisation of the cells (division of labour) similar to social insects experiencing queuing and bottleneck situations. A lot of projects are part of the Hydra Project, like M-TRAN, Swarm-bots or HYDRON.

2.7.1 HYDRON

A Hydron unit is a roughly circular robot (Figure 2.6) intended for use in collective tasks under water. It is is actuated in the horizontal plane by four nozzles which expel water drawn through an impellor at the bottom of the unit, with a rotating collar selecting the active nozzle. A syringe draws or expels water through the bottom of the unit to control unit buoyancy, actuating the unit along the vertical axis.
This modular robot is the most similar to our Neubot although there are important differences between the "philosophy" of the projects. We will show in detail these differences in the next chapter.

Figure 2.6 : *The final version of the hydron, containing all mechanical and electronic systems, including optical communications.*

2.8 Lattice robots

Lattice robots are also modular robots but are based on a very different architectures. Indeed, reconfiguration is the only capability of a lattice robot, locomotion is thus achieve thanks to it and is not, like in the chain robots only a way to be

adaptive. Amongst the most important projects, we find *Telecube* [13], *Crystalline* [14] or *ATRON* [15] which is part of the Hydra Project.

2.9 Adam

Adam is not a modular robot but a system to evolve and simulate modular robots. In the first version, the simulated robots was not reconfigurable and loop configuration was not allowed due to the tree genotype. Daniel Marbach used Adam to co-evolve configuration and control as a test of his simulator [5].

Chapter 3

Neubot Project

The Neubot Project is developed by the LSL in collaboration with the ASL and with the aim of developing a new type of modular robot. Heterogeneity is the main characteristic of this new modular robot, i.e. the modules which make it up are not all similar but show different characteristics. This means the possibility for a module to "recruit" others modules depending on their type and its own needs. It must also (eventually) take the behaviour of the others into account in order to lead to a coherent group behaviour towards the wanted aim. Various fields of study are thus shown, for example, the behaviour of individuals with a view to reaching given heterogen structures or the definition of several types of modules in a given situation.

The other main characteristic of this project is the environment in which the modules will be : the water. Several challenges appear with this choice: the three-dimensionality which is a big problem, but also from the point of view of the material realisation the necessity to create totally waterproof modules able to float below the surface without spending energy. In the future, it is envisaged that the modules will be both on the ground and underwater thanks to, among others, new types of modules for switching between the two environments. Generally speaking, the modules have the possibility to move themselves and are able to attach together in order to form complex structures while being completely autonomous when they are alone. They must also be able to communicate even in a very basic way, in order to be able to coordinate between them.

3.1 The Basic Module

It were necessary to develop a first basic module before the specialized modules. We will first see the specification of this module before describing it precisely and showing its particularities.

3.1.1 Specifications

The following specifications were required. For more details about the design of the module refer to [33].

1. Required
    - Fully autonomous underwater operation
    - Reconfiguration capability
    - Mobility
    - Smaller than 1dm3
2. Desired
    - Selective self-assembly
    - Energy sharing
    - Homogeneity (for this basic module only)
    - Unisex connectivity
    - Unconstrained 3-axis mobility
    - Encapsulation
    - Distributed/redundant subsystems

3.1.2 Description

The specifications described within section 3.1.1 leads to the module represented on figure 3.1. Contrary to all the projects presented in chapter 2 except HYDRON project (not taking into account the lattice modular robotic systems), the Neubot module is made of only one block providing 6 square docking faces and 8 water jets. Only the links with the other modules enable it to create a joint actuated by a motor. This joint is parallel to the normal of the linked surface. This type of joint has the advantage on a standard hinge joint which allow six degrees of freedom instead of four. However, the four nonrotary degrees of freedom requires more modules to be realized by parallel to the normal joints, causing an additional cost for this type of movement. This point will be discussed again in the chapter on the motivations for the co-evolution.

Connection and actuation of joints The connection between the modules is based on permanent magnetic dipoles. Two articulated rings of magnetic dipoles (as shown in Figure 3.2) establish the connection between the modules and create an actuated joint with one rotational degree of freedom. The magnetic polarity of the inner and outer ring can be adjusted to establish a neutral, attractive

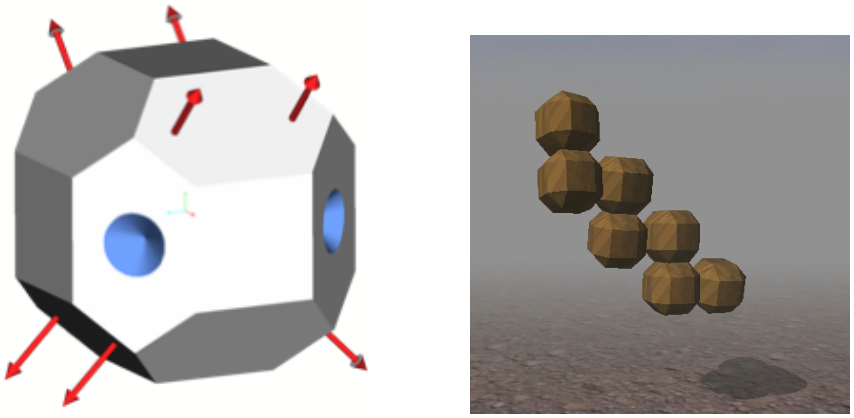

Figure 3.1 : *The basic neubot module. Left: On the module, we can see the square docking faces. The water inlet valves are marked by the blue circles and the red arrows show the 8 water jets (Picture by Tiago Bertolote). Right: A structure of modules, i.e. a robot.*

or repulsive effect between two modules. The connectors thus do not consume any energy for creating and maintaining joints and are encapsulated within the module. These connectors can be put in rotation thus making the two attached modules turn. The detachment is done by using repulsive forces.

Figure 3.2 : *The system for interconnecting the modules using an articulated ring of magnetic dipoles from one squared face (this device will be placed in the module). The magnetic polarity of the inner and outer part can be changed in order to establish an attractive or repulsive effect between two modules.*

Propulsion The propulsion of the modules when they are alone is ensured by eight water jets. As the modules are ballasted, they have four degrees of freedom for the alone locomotion. The red arrows on figure 3.1 shows some of the valves through which the water is expulsed.

Communication and sensors The type of sensors with which the modules will be equipped for detecting obstacles or other modules and for communicating is still under discussion but two options are being considered: on one hand a vision based system like photo-receptors (or a camera) and leds allowing at the same time to roughly make out obstacles and to communicate. On the other hand, an electro-sensors system able to make out obstacles and other modules more precisely (Figure 3.3) as well as to communicate varying its own magnetic field. This second option is much more neat and interesting but requires more research and work than the first. The first prototypes will thus be probably equipped by a vision-based system.

Furthermore, all actuated joints i.e. the magnetic dipole rings are sensed and the modules have a pressure sensor allowing a rough estimation of depth.

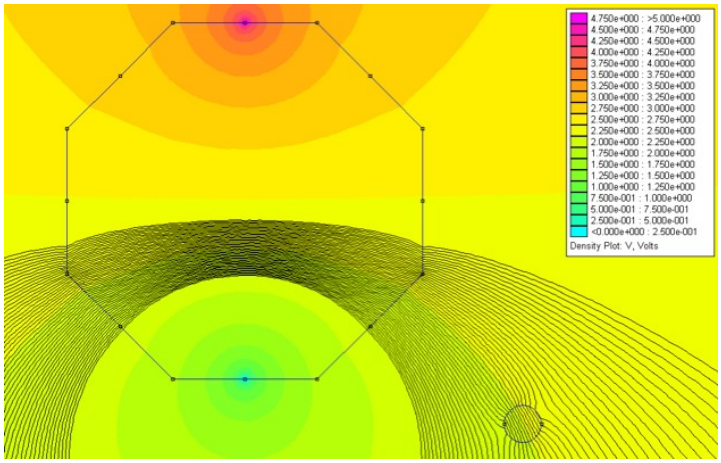

Figure 3.3 : *A simulation of the electro-sensors.*

3.2 Particularities

Compared to the projects described in chapter 2, the Neubot project is characterized by several points. Firstly, the type of joints choosen whose axis are parallel to the normal of the face, was not really studied until now and the structures being able to benefit from this characteristic are for the moment little known. By definition, only this type of joint makes it possible to have six degrees of freedom and thus the maximum latitude as for the usable structures. Then, the autonomy of the modules as well from the point of view of attachment as of the locomotion enables them to configure and reconfigure without human assistance. We can thus plan thorough strategies of reconfigurations and repairs, for example, rejecting a broken module and its replacement by another independent of the structure. Finally, the aquatic environment offers new challenges as well, in terms of material manufacture as well as individual and group strategies and behavior.

Part I

A Framework for Simulation of Modular Robots

Chapter 4

Tools

This chapter describes some of the tools used for this project. The framework was developed in C++ on the three main operating sytems : Windows, MacOS X and Linux. As we know, the framework has been tested successfully on the three platforms.

4.1 Open Dynamics Engine

All the physical simulations were exclusively done with the *Open Dynamics Engine* (ODE) version 0.5 [34]. ODE is an open source, high performance library for simulating rigid body dynamics. It is relatively stable and accurate (for an under-development software) and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction.

4.2 Picasso

Picasso was developed by Daniel Marbach during his Semester Project [28]. This library is used as the graphical part of the physical simulation. ODE provides one named "drawstuff" but it is very simple. Picasso uses GLUT and offers more capabilities and is simple to extend.

4.3 Adam and others simulators

Two softwares were already developed by other students of the BIRG to simulate modular robot. Bertrand Mesot was the author of one of them [31], which is relatively simple and essentially developed for a basic cubic module. The second one, named Adam, is developed by Daniel Marbach. Adam is the base/father

of our own simulator however we have all re-coded in order to have a clean-cut architecture and code. His design was a source of inspiration for us.

4.4 Softwares

We programmed with Visual C++ on Windows, XCode on MacOS X and a simple text editor with g++ 3 on Linux.

Chapter 5

Design

The purpose of our Framework was to simplify the development of simulation of modular robots. It must be generalistic, efficient and reusable but also easy to use. So the result is the design draw on the figure 5.1.

The two central classes are *Module* and *Simulation*. The first one is abstract and must be sub-classed for creating a new type of module. The second one is a singleton which deals with the ODE world (creation, stepping) and handle the collision detection with dedicated callback functions. A module has one and only one controller which must be sub-class of the abstract class *Controller*. This controller will at each step calculate the new position of the motor(s). Finally a module can have some sensors that will be encapsulated into a *Sensors* object which simply contains the values of the sensors. These values must be updated at each step by the module or the simulation or anything else.

The *Serializer* abstract class offers the possibility to read/write from a file the structure of the whole robot. As the information needed depends strongly on the module, it is necessary to write a specific sub-class for each type of module.

Several classes offer some useful services like the capabilities of loading from an *.obj* file[1] the shape of the module (it will be created using *Trimesh*[2]) supplied by *GeometryLoader* or a lot of functions for the matrix calculations or the I/O from/to a file supplied by the *Utilities* class. The *Settings* class offers the possibility to read parameters from a file needed by the application using a very simple grammar (for more informations about the syntax of this file and about the details of the framework see Appendix A).

[1]The OBJ file format by Wavefront is a simple standard format for 3D geometry.
[2]A triangle mesh (TriMesh) represents an arbitrary collection of triangles in ODE.

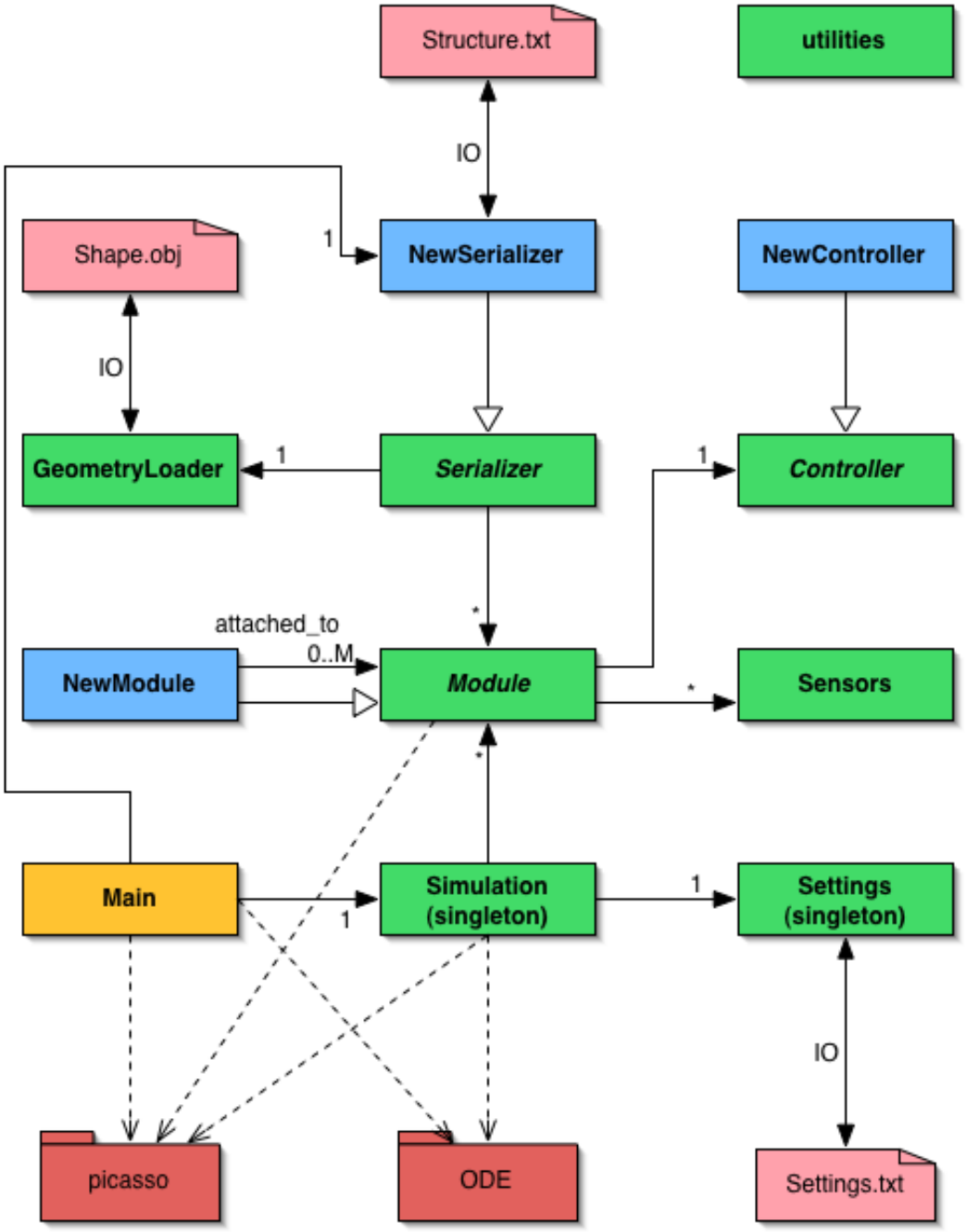

Figure 5.1 : *The design of the framework. Green boxes : framework's classes. Blue boxes : examples for a new module. Pink boxes : files used by the software. Dark pink boxes : packages.*

Chapter 6

Neubot Simulation using the Framework

Simulation plays an important role in evolutionary robotics because it is not possible, in general, to calculate analytically the fitness. Therefore, the fitness has to be evaluated either on the real robot (online) or in simulation (offline). Although the online evaluation has several advantages (*wysigwyg* by example), its slowness is too big a problem to envisage the use of it. In addition, as the physical realization of the neubot module was made concurrently with the simulation, it was useful to make some preliminary tests in the simulation to help those who deal with the hardware. As a result, the basic Neubot module has been implemented into the previously described framework.

6.1 Degree of Details of the Neubot Modules

The main question during the implementation of the Neubot module was to choose between a high degree of precision or a fast but unreliable simulation. The risk was to develop a controller or to verify an hypothesis in the simulation but which would be completely useless in the real world. To ensure a good transfer from simulation to real world the module and the environment must be carefully, but not necesserily accurately, reproduced. However, it is a challenge to achieve a simulator without having the real module. Therefore, as soon as the real module is available, all the coefficients and variables must be determined experimentally and changed if necessary.

We had to make a lot of choices and hypotheses to decide which features would be relevant or not, which details have to be accurately modeled or not. The following sections explain the implementation and the decisions that we made.

6.2 Shape of the module

In the first phase we developed a cubic module with proeminent joints (Figure 6.1). The joints had to be proeminent because otherwise the modules can't rotate correctly. Although this shape doesn't pose a problem with given robots structure, its behaviour when two lone modules come near each other was wrong. The modules indeed was sometimes unable to attach together because the proeminent joints block them. Therefore, we decided to create the real shape using *Trimesh* in ODE. A triangle mesh (TriMesh) represents an arbitrary collection of triangles. This feature of ODE is new and has several "teething" problems with the collision detection. Thus we decided to put three solid cubes into the module to be sure that they will not intersect together. This option requires a little bit more calculation time but avoids the necessity to entirely rewrite a collision detection engine.

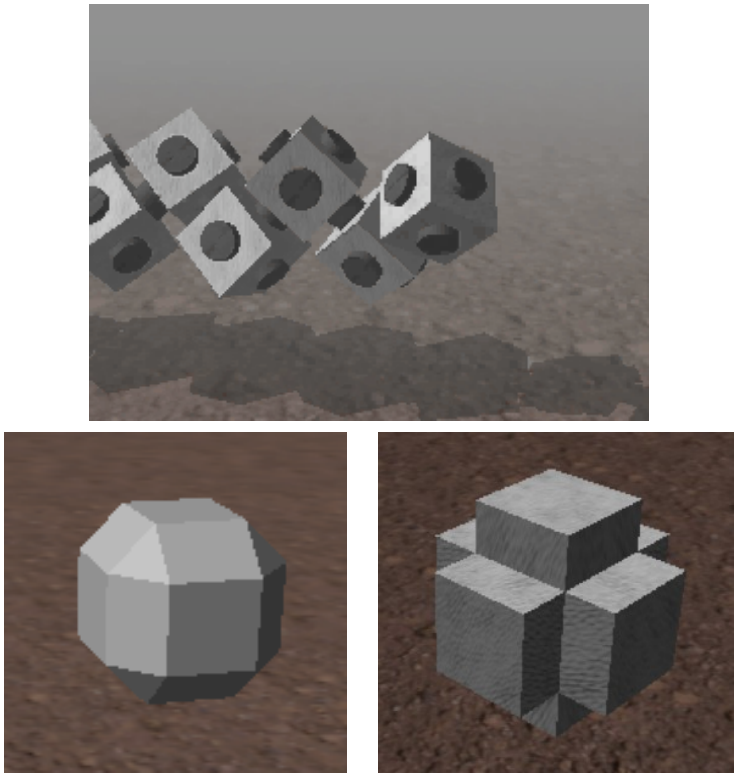

Figure 6.1 : *Top: the old shape of the modules into the simulation (above assembled). Left: the new shape. Right: the three solid cubes (they intersect) appear if we remove the Trimesh envelop.*

6.3 Physic Model of the Water

The Neubot Module is an underwater device and we had to develop a physical model of water. The fluid dynamics is an excessively complex field and it quickly appeared that it would not be possible to have an accurate model. The chosen model is based on two very strong simplifications

- Only three classes of forces act on the model : *Buoyancy Forces*, *Inertial Drag Forces* and *Viscous Drag Forces*.
- The above forces can be split up in the principal axis of the module

These simplifications are very strong and very dangerous and it is possible that the "Reality Gap" becomes very high. However, this model allows a very fast simulation of the system.

As the theoretical part of the fluid dynamics was essentially made by Peter Dürr, only a simplified explaination is provided here. Please refer to [32] for a more detailed description of the model.

The hydrodynamic drag forces exerted on the module by the fluid (forces made for the contrary sense of trip) during a translation are

$$F_T = -\frac{1}{2}\varrho_{water} \cdot L \cdot C_D \cdot v^2$$

where L is the frontal area of the object in the fluid, ϱ_{water} is the fluid density , C_D the (velocity dpendent) drag coefficient and v is the velocity of the module (because we decided that the fluid has no velocity).

The forces exerted on the module by the fluid during a rotation in order to "struggle" against them are visquous and are implemented as

$$F_R = L \cdot C_T \cdot v^2$$

where L is the frontal area of the object in the fluid, C_T the dimensionless rotation friction constant and v is the velocity of the module (because we decided that the fluid has no velocity).
Finally, we had the Buoyant force defined as follow

$$F_B = \varrho_{water} g V_{module}$$

where ϱ_{water} represents the density of the fluid, g is gravitation acceleration and V_{module} the volume of the module. As the modeled module has a density that is about equal to the density of the surrounding water ϱ_{water}, the effect of the buoyant force and gravity compensate each other.

We must underline some remarks about the simplifications that we made. First, the modules are simplified as cubes as regards to the fluid dynamics (not for the collisions). Then, the modules are taken independently from each others for the calculation of the forces and not as a structure. Indeed, it would have been too complicated to calculate the water's dynamics around an assembled modular robot. However, the faces on which another module is attached are not taken into account for the calculation of the forces. Moreover, the faces themselves are taken independently from each others at the time of this calculation. Then, resistance against rotation is independent from inertia (i.e. we use only the visquous model) what requires to be checked when a prototype of module will be available. All the values of the variables as C_T or C_D have to be also checked. Finally, the μ (visquosity) is the same whatever the direction of the face.

Another possibility of simplification would be to consider the modules as - spheres. We thought it would be more simple to choose the first possibility because with cube we can calculate the forces on each empty face, although with a sphere it would not be feasable.

Peter Dürr have later developed a physic model using an approximagtion of the modules as spheres. As all the first tests and simulations of the second part of this project have been made using the cubic model, I continued to use this one during whole project.

6.4 Attachment, Magnets and Joints

We have modeled the magnetic dipoles of each face of the module by a sphere representing the limit beyond which the effect of a magnet pair is neglected (Fig. 6.2). This is important for reducing the overall complexity and also the number of forces to calculate. The problem of calculating the forces and torques which appear between two modules due to the magnets can be reduced to the calculation of the forces between each pair of magnetic rings.

First of all, a different approach had been considered in which there were 3 dipoles on each face that we had represented by 6 spheres. However the first simulations which we made with this type of interconnection showed instabilities which were confirmed by real tests. The system thus evolved and the implementation of the new type of interconnection was made by Peter Dürr on the basis of what we had done jointly before.

When the spheres of two different modules intersects, we calculate the vector v_1 between the centers of the spheres and the force

$$F_{magnet} = \frac{1}{\mu}\frac{C_1 C_2}{d^2}$$

where μ is the magnetic permeability of water which is very close to the magnetic

Figure 6.2 : *Left: simply the module. Middle: Old magnetic system. On each face appears six spheres. The positives are red, the negatives black. The dipoles seem to be one bicolor sphere because of the proximity of the two magnets. Right: the next version of the magnetic attachment system.*

permeability of free space ($\mu_0 = 4\pi \cdot 10^{-7}[\frac{N}{A^2}]$), C_1 and C_2 are the magnetic force of each magnet (here 9 [N]) and d is the distance between the magnets. We apply this force on the two modules in direction of v_1 and in the sense for getting closer.

When two modules are close enough to each other, and that the alignment is correct, we create an ODE joint between them and we remove the magnetic spheres of the concerning faces. The magnet's spheres are not attached with a joint to their module. We calculate at each step the new position of each sphere. We estimated that this calculation although it is relatively long, required less calculation time than ODE joints. Indeed, the joints in ODE require also the calculation of constraints on them - an exercise which would be entirely unnecessary for us.

The decoupling is done by changing the polarity of the rings of one module so that the magnet forces become repulsive ([33]) and we destroy the ODE joint between the two modules. In case of a too strong force which separate the modules, we break the joint ourselves because ODE doesn't offer a practical and neat way to break a joint if the forces are excessive.

6.5 Sensors

Any kind of sensors can be easily added to the modules in the simulator. The first one is placed in the actuated joints, i.e. the magnetic dipoles, in order to know its position. It is necessary for regulating the motors amongst others. Peter Dürr have implemented others sensors for the purposes of its own project [32].

6.6 Motors

We implemented the actuators of the rotational motors with a PID controller instead of using the simple angular motor provided by ODE because it is not well documented and we did not know how it is really implemented. Our PID controller is thus implemented as follow

$$F = C \cdot \Big[- K_P(\vartheta - \tilde{\vartheta}) - K_D\dot{\vartheta} + K_I \int (\vartheta - \tilde{\vartheta})dt \Big]$$

The different parameters K_P, K_I, K_D and the force multiplicator C had to be evaluated manually (the values of the different parameters of the simulator used during the second part of this project are all referenced in Table A.4).

Part II

Self-Organisation of Locomotion under Water

Chapter 7

Introduction

One of the fundamental aspects of a robot is its locomotion ability. If we hope that a robot can act in an autonomous way it is crucial in most cases that it can move in a fast and precise way. However, this capacity remains a challenge and many strategies were elaborated by scientists to solve this problem. A promising way is provided by the observation of the nature and is known as *Central Pattern Generator* about which we will speak again in detail in chapter 9.

In modular robotics, an additional problem appears. As the shape of the robot is free and flexible, it is restrictive to predefine it. The Neubot project, with the particular shape of its modules and the parallel to the normal rotation axis, presents an additional difficulty. Indeed the human brain is not familiar with this type of movement which is quasi-unknown in nature and because of this it is difficult to imagine effective forms for locomotion. All these reasons make it necessary to evolve not only the controller or the parameters regulating the gait but also the shape of the robots. The co-evolution thus appears as a suitable solution to find at the same time forms and controllers adapted to the locomotion. However, initially, several robots whose form was defined by us were evolved to test the relevance of our approach based on neural networks and synaptic spreading.

7.1 Tools

GAlib Genetic Algorithm were done with GAlib, an object oriented and highly parameterisable genetic algorithm library. GAlib implements the most common GAs, types of genome, crossover and mutation functions. If nothing corresponds to the needs of the programmer, he can program his own sub-classes easily thanks to the high extensibility of this library [35].

GSL The nonlinear equations integrations were done with the GNU Scientific Library (GSL). This library is a collection of routines for numerical computing which provides a well-defined C language Applications Programming Interface (API) for common numerical functions.

Debugging and profiling Developing a program is not only writing code but also debugging and optimising this code. To do that, I used the debugger included in *XCode* based on *GDB*, *LeakTracer*[1] and *Valgrind*[2] for tracking the memory leaks and *gprof* for the profiling.

[1]http://www.andreasen.org/LeakTracer/
[2]http://valgrind.kde.org/

Chapter 8

Firsts tests

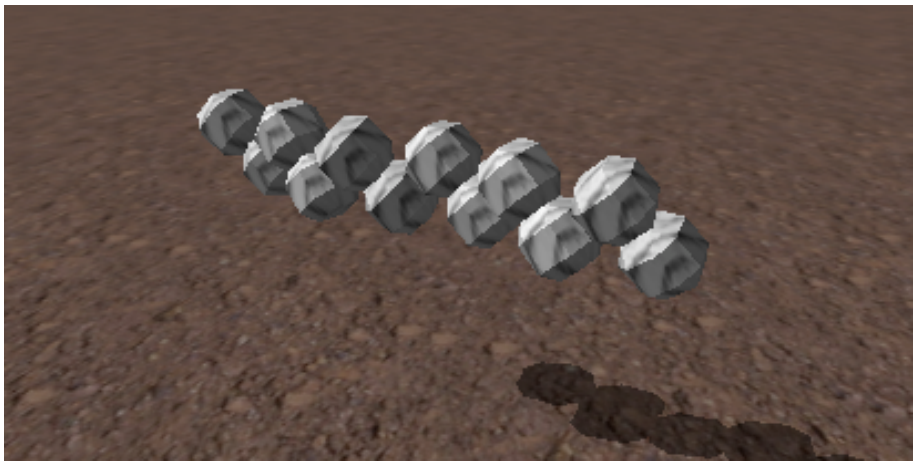

Figure 8.1 : *This "snake" simply use sinus as controller with a dephasage between the modules.*

As a first test, I tried to use simple sinus as controller and to evolve his parameters. For simple forms, a snake for example, the result was good (Figure 8.1), but this seems to be a dead end. Indeed, the sine-based controller are not flexible enough. If we hope to realize several gaits with the same structure, we must think about the transitions between the gaits. These transitions imply abrupt changes of frequency, amplitude and/or phase but abrupt changes of parameters leads to abrupt modifications of the setpoints. Therefore, the gait transitions will be unnatural, in nature the gait transitions always occur smoothly, and unapplicable because of the abrupt change of power applied on the motors. Furthermore, with sinusoidal signals, there is no simple way to incorporate sensory feedback.

Chapter 9

CPG's and Oscillators

As said in the introduction, the CPGs are neural networks that generate oscillatory signals from a tonic input coming from the brain. These signals command the muscles. The oscillations cannot be simplified as simple sinusoidal signals because they are not flexible enough. Indeed, gait transitions need the possibility of changing the phases, amplitude and frequencies of the sinusoidal signal but without brutal changes of the signal. Thus, the oscillators must enable the possibility of smooth transition. Moreover, they must be resistant to perturbation and offers a simple way to incorporate sensory feedback.

The CPGs can be implemented at different levels of details. A common way used at BIRG is to introduce non-linear oscillators as mathematical models of the natural CPGs [4] [30]. On the other hand, the implementation of a neural network as oscillation generator is closer to the biological reality. Both have the characteristic wanted (smooth transitions, resistant, sensory feedback). In the next section, I will quickly recall the non-linear oscillators way, but the final solution chosen is a neural network. Indeed, the other parts of the controller of the Neubot module (reconfiguration) will be realized thanks to neural networks [32] and thus with the intention of being totally coherent we want that all the part of this controller be at the same level of details. Furthermore, the oscillations generated by a neural network are more interesting because of the further complexity they offer. Finally, this approach is more "biologically inspired".

9.1 Non-Linear Oscillators

I explored briefly the use of non-linear oscillators instead of simple sinus or artificial neural network. I developed a few classes (Figure B.1 in Appendix B) and I tested them. The oscillators implement the following non-linear oscillator [4][30]:

$$\tau \dot{v_i} = -\alpha \frac{x_i^2 + v_i^2 - E}{E} v_i - x_i + \sum_j^N (a_{ij} x_j + b_{ij} v_j)$$

$$\tau \dot{x_i} = v_i$$

The results of this approach with a simple GA that optimize the oscillation amplitude (E), oscillation central angle (x_0) and the coupling strength (a_{ij} and b_{ij}) have been poor. I think this is essentially because of the combination of several factors: the use of predefined forms that were not well formed for the locomotion with the Neubot module, the short time passed on this option and a too basic fitness function. This approach would have been however a good idea, but as said above we want coherence between the different part of the controller and we choose therefore to generate oscillations through an artificial neural network described in the next section.

9.2 Oscillations through Neural Network

As previously said, the biologically inspired locomotion with CPG's need oscillators and we want that these oscillations be generated by a neural network. Thus the oscillator is developed by evolving the connectivity and the neuron parameters of an eight-neuron neural network. This approach is inspired by the work of Auke Ijspeert described in [3]. The design diagram of the implementation can be found in Figure B.2 in Appendix B.

9.2.1 Neuron model

The oscillators are composed of neuron units of intermediate complexity between abstract binary neurons used traditionnally in artificial neural networks and detailed models used in computational neuroscience. A neuron unit is modeled as a leaky integrator. According to this model, the mean membrane potential m_i and the short-term average firing frequency x_i of a neuron i are governed by the equation:

$$\tau \frac{\mathrm{d}m_i}{\mathrm{d}t} = -m_i + \sum_j w_{i,j} x_j$$

$$x_i = (1 + e^{-(m_i + b_i)})^{-1}$$

where b_i is the neuron's bias, τ_i is a time constant associated with the passive properties of the neuron's membrane, and $w_{i,j}$ is the synaptic weight of a connection from neuron j to neuron i.

9.2.2 Genetic Algorithm

GALib [35] was used as the base of the genetic algorithm (GA). The parameters of a neural network are encoded into chromosome which are fixed-length strings of real values (instead of the traditional binary encoding). The GA gives a fitness value to each chromosome (individual) according to a fitness function that

determines the good appropriateness between the output of the neural network and the awaited oscillatory behavior.

The GA begins with a population of N individual randomly created. At each generation, the crossover, mutation and pruning operators are applied for creating C new children. The crossover operator takes two parents and for each position exchange the genes at that position with probability P_C. Mutation randomly selects a position into the chromosome with a probability P_M and changes the corresponding value according to a a Gaussian distribution around the old value. The pruning operator is specific to the neural network optimisation and prunes a connection (probability P_P) by setting the gene corresponding to the weight of this connection to zero. The children are then evaluated and, as the population is fixed, the C worst individuals are rejected from the total population (parents and children). The parameters used are given in Table 9.1.

	N	C	P_M	P_C	P_P
Value	100	50	0.4	0.5	0.05

Table 9.1 : *Parameters of the genetic algorithm.*

9.2.3 Encoding and fitness function

A network is encoded into 43 genes. The network assumes to have a left-right symetry (cf Fig. 9.1) with three neurons and one motoneuron on each side. The motoneuron is the one which is evaluated. For each type of neuron, a genome encodes its time constant (τ), its bias (β), its sign (to know if it is excitatory (+) or inhibitory (-)) and the synaptic weights of the outwards connections from itself to the other neurons types and to itself (32 genes). A chromosome also encodes the τ and β of motoneuron and the weights of the connections from the BS to the eight types of neurons (11 genes). The parameters are encoded into the genes with the following boundary values: 20.0 and 500.0 ms for the time constants, -10.0 and 10.0 for the biases, and 0.0 and 10.0 for the synaptic weights.

The fitness function is defined to reward solutions that oscillate regularly with left and right motoneurons out-of-phase, whose frequency and amplitude of oscillation can be varied with the level of BS (increasing the tonic drive BS must increase monotonically both the amplitude and the frequency) and whose number of connections are as few as possible. Mathematically, the fitness function is a product of six factors:

$$fit = fit\_nb\_connect \cdot fit\_oscil \cdot fit\_regularity \cdot fit\_oscil\_phase \cdot fit\_freq\_range \cdot fit\_ampl$$

Each factor varies linearly between 0.05 and 1.0. Table 9.2 summarize how each factor is calculated using a variable which varies between a bad and a good

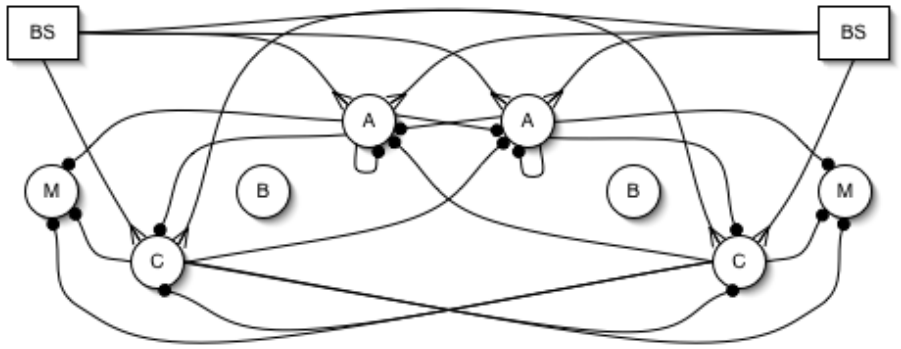

Figure 9.1 : *The oscillator : neural network and connectivity. A, B and C are three types of neurons and M are motoneurons. BS is the input from the brainstem.*

boundary. Thus we do a mapping of the range of the variable to the range of the factors. The target of the network is specified by the good boundaries of the variables.

Function	Variable	Bounds [*bad*, *good*]
fit_nb_connect	number of connections	[24, 8]
fit_oscil	number of optima	[0, 20]
fit_regularity	SD of periods	[0.25, 0.0]
fit_oscil_phase	$\lvert\theta_{oscil} - 0.5\rvert$	[0.5, 0.0]
fit_freq_range	max_freq / min_freq	[1.0, 15.0]
fit_ampl	mean of amplitude	[0, 0.1]

Table 9.2 : *Variables and boundaries for the fitness function of the neural network oscillator. If a variable has a better value than the good bound, the corresponding fitness factor will be 1. fit_nb_connect counts the connection from a neuron of type A, B or C to any other including motoneurons.* θ_{oscil} *is the phase between the left and right motoneuron. As it is measured as a ratio of the cycle duration, 0.5 means anti-phase. The frequency range is optimized by optimizing the ratio between maximum and minimum frequencies with min_freq* $\geq$ *0.8*

.

The fitness function is a product rather than a sum because this solution ensures that all the six factors will be optimized and not only one or two that would give a great fitness with a sum. The *fit_ampl* factor has firstly not been used but several solutions have then very little amplitude. So this factor ensures that at least the amplitude is 0.1. Because of the future use of this oscillator into evolved robot with one gene for each connection in each module, the number of connections must be as few as possible. The *fit_nb_connect* factor limits this number. In order to determine the range of frequencies at which the networks can oscillate, a first evaluation (12 seconds) is made with a level of tonic drive

of one. If oscillations appear at this level, we decrease it of 0.1 and retest (same duration) until oscillations disapear. The same operations are done increasing the BS in order to get the maximum frequency.

	M_l	A_l	B_l	C_l	BS_l
τ	234 ms	37 ms	280 ms	346 ms	X
β	8.64	1.4	-10	-3.07	X
sign	X	-1	1	-1	X
M_l	X	4.4	-	10	-
A_l	X	1.4	-	-	2.8
B_l	X	-	-	-	-
C_l	X	10	-	-	1.4
M_r	X	-	-	1.3	-
A_r	X	5.4	-	9.9	3
B_r	X	-	-	-	-
C_r	X	-	-	10	10

Table 9.3 : *The genome of the best oscillator. The three first lines of the table gives the parameters of each type of neuron (tau, bias and sign). The lower part of the table gives the synaptic weights for all connections from M_l, A_l, B_l and C_l (the neurons on the left side) to each other neuron. The encoding deals with the symetry of the connections of the neural network and avoid to repeat same information. 'X' specifies that no value is needed here and '-' means that this weight is null.*

9.2.4 Results

A first set of five runs was started without the fitness factors $fit\_nb\_connect$ and fit_ampl. It appears that in three runs the amplitude becomes very little (e.g. approximatively 10^{-2}) but the general behaviour is correct. The two other were correct but the maximum frequency was limited to 4 Hz, which is the half of the best result described in [3]. Five new runs were carried out with the limiting factor fit_ampl. They all converge to networks able to generate regular oscillations whose frequency can be modulated using the tonic drive (BS). One of them, the best, had the same topology as the network found by Auke Ijspeert previously cited, but with small differences on the weights and thus on the neural activity. As the future step of the project was to evolve the robot's controller for locomotion with one gene for each connection of each oscillator, it was very important that the number of internal connections (from a neuron of type A, B or C to neurons of type A, B, C or M) was as little as possible. Therefore, five runs were launched again with the added fitness factor $fit\_nb\_connect$. All runs converged to correct networks with the wanted behaviour, but with a smaller range of frequencies. All the other fitness factors were as good as in the previous runs carried out.

Figure 9.1 and Table 9.3 give respectively the topology and the parameters of the best evolved network of "third generation". The neurons of type B are not yet connected to any other neuron and become useless. The total number of internal connections is 8 instead of 14 for the absolute best one found during the second set of runs. The level of tonic drive (BS) can be varied between 0.2 and 1.13 (the frequency and the amplitude increase monotonically in this range of input). A range of frequency from 0.8 to 3.3 Hz can thus be covered and this, in fact, is three times lesser than the best one of the second set (Fig. 9.4). When the tonic input is increased above the upper limit of the monotonic range, the frequency decreases monotonically until 0.75 before going abruptly to zero. The graph of frequencies is quasi symetrique around 1.13. All the networks found during the fifteen runs have this quasi-symetry more or less.

The oscillations are regular (Fig. 9.2) and the same whatever are the initial conditions of the network (if we run several times the network with random values as parameters, it converges every time to the same oscillations but with a more or less time of stabilisation). Thus a stable and unique limit cycle can be drawn (Fig. 9.3) which is the cause of the oscillations and which proves the resistance of our oscillator. Because of the negative biases of neurons A and C (B is not used) the network is not oscillating when there is no input (none of the neurons of the network produces an output, then the network is quiet).

9.2.5 Discussion

The best network found during the second set of runs, is more or less the same as the one found by Auke Ijspeert. It is really better than the others and then I think this is a very good but little and difficult to find optimum. A lot of local optima with medium performances have been discovered by the GA (all runs converged to different solutions) thus the fitness landscape must be relatively plane with a lot of local attractors.

The choice between a good and expansive network, in term of connections, and a poorer network with fewer connections is not easy. Because of the probable great number of parameters necessary in the next step of the project, I opt for the second one which will save precious computation time. Moreover, the modules will have physical and mechanical limits which let me think that a frequency greater than three Herz is not realistic in the first prototypes.

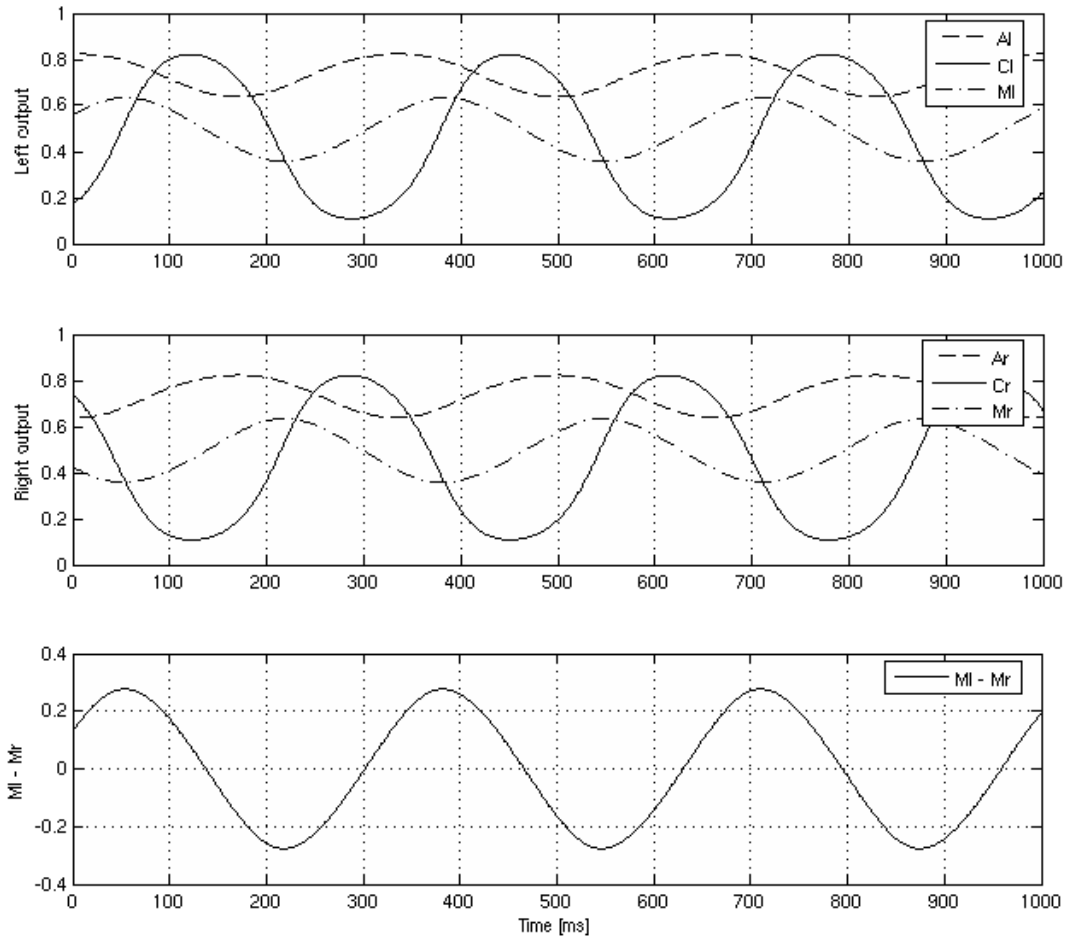

Figure 9.2 : *Neural activity in the oscillator. The upper graph shows the activity of the left side of the network, the middle the right side and the lower the difference between output of Ml and Mr.*

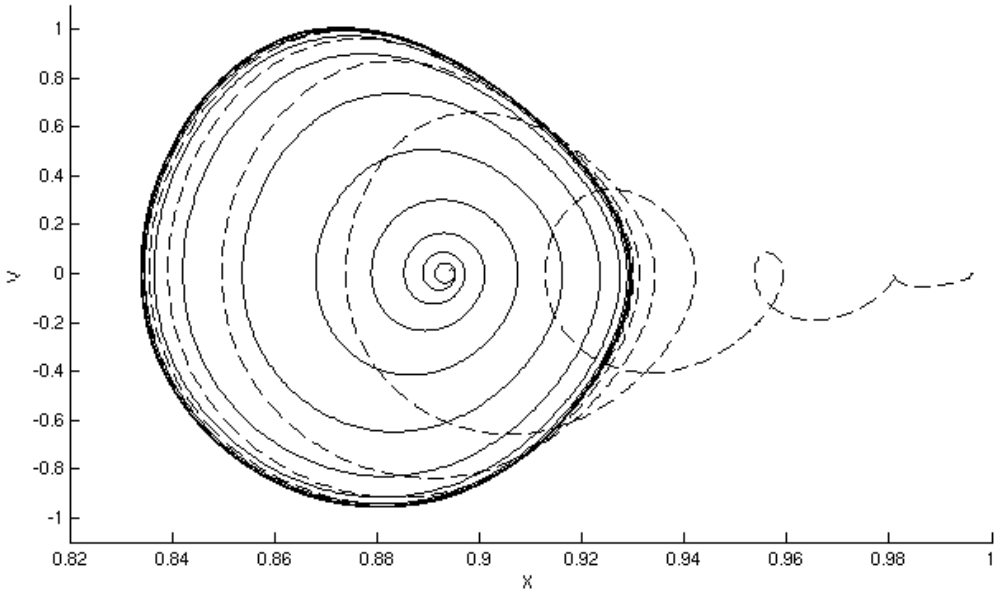

Figure 9.3 : *Limit cycle of a standalone oscillator. For more legibility, only two oscillations started with different values are shown above, but a test with 100 different initial values confirms that all the oscillations converge to the same circular attractor (even if the membrane off all the eight neurons are set to zero at start).*

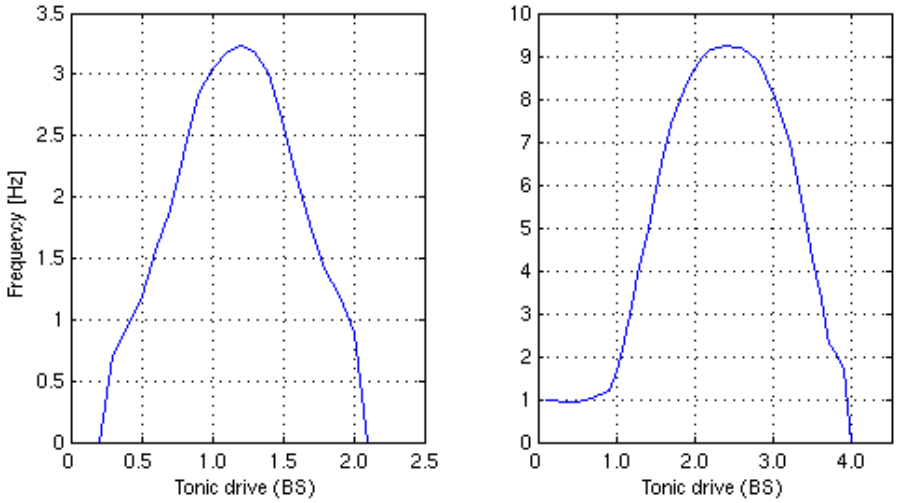

Figure 9.4 : *The frequency graphes of the two best oscillators. Left : the frequency vs BS of the oscillator with 6 neurons (best of the third set of runs). Right : the frequency vs BS of the best oscillator found during second set of runs. This one is three times better than the one on the left side.*

Chapter 10

Locomotion using predefined structure

The next step in order to give locomotion habitility to a modular robot is to use the oscillator developped in the previous chapter. The aim of this stage is to develop a complete CPG by coupling the oscillators together. As a first test and before using co-evolution, these couplings were evolved into predefined structures which will be described in the following section. This stage is destined to confirm that the system of coupling discovered in the lamprey is applicable to our modular robots despite the rotation axis which is parallel to the normal.

The coupling between segments of the lamprey is obtained using synaptic spreading. The idea is to project the connection between two neurons of an oscillator to corresponding neurons in neighboring segments [26]. The synaptic spreading can be to the nearest neighbor segment only or even further. In our model, projections are limited to different values dependent on the structure used and are more or less proportionnal to the number of oscillator in the robot.

The design diagram of the classes developed is shown on Figure B.2 in Appendix B. A new Serializer/Deserializer has been implemented and a new controller too. The detailed description of the classes is in A.3. The different gait describes in this chapter can be viewed in little movies on the website of this project[1] [36].

One last important remark on the implementation: the magnets have been deactivated for avoiding their influences and saving computation time. As the new system of attachment (cf section 6.4) allows to deactivate the magnets of a face, our premiss is thus realistic. The joints are even breakable if the forces are too strong.

[1] http://birg.epfl.ch/page56704.html

10.1 Structures

Three structures have been defined : a snake, a tetra and an arrow (Fig 10.1). Firstly a fourth structure was designed, a spiral, but, as we will explain in detail later, its results are similar as those of the snake. Each joint of a robot can be actuated or not and if actuated it is only by one of the two motors (each face of the modules has a motor, thus each joint has two motors). Indeed, having two oscillators by joint would be too complex and would double the number of parameters to evolve. Another possibility would be to use the same oscillator for the two motors of the joint and then to double the power of the oscillator. As the simple oscillator seems to be sufficient for actuating the robots, I decided to remain with only one motor actuated.

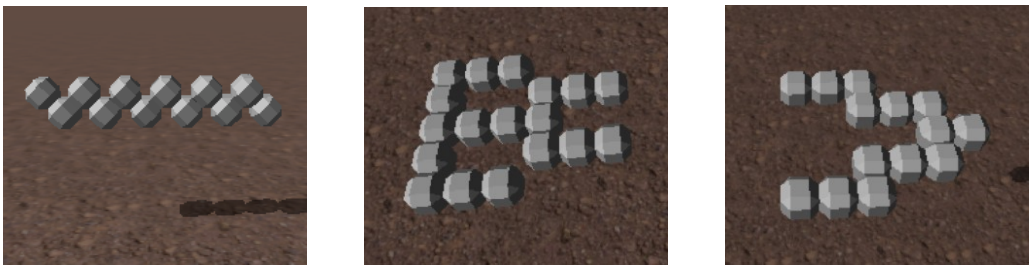

Figure 10.1 : *Predefined structures. Left : a snake (its shape is not a simple alignment of modules because of the particular rotation axis). Middle : a tetrapode. Right : an "arrow".*

The joints don't need to be all actuated and therefore the choice of which ones are fixed or not must be carefully done. For the snake, they are all actuated. In the tetrapode, only six joints are not fixed (Fig. 10.2). Firstly, two oscillators were put in the "body", but they were very unstable in the simulator because of the too strong force needed to rotate the upper and lower part of the body. Finally, the arrow has a chain of oscillator along the two sides.

When the oscillators are placed, the connections between them has to be discussed. I opt for an oscillator's chain for the snake and the arrow. In this one, it is important that the two sides are connected for their synchronisation. The tetrapode is less coarse and I decided to use the same coupling as the quadruped in [30].

The oscillator used is the neural network developed in the previous chapter. Instead of using one of the two motoneurons, I decided to use the difference between the two motoneurons in order to have an oscillation centered into zero (i.e. with a null offset) as shown on figure 9.2 page 43.

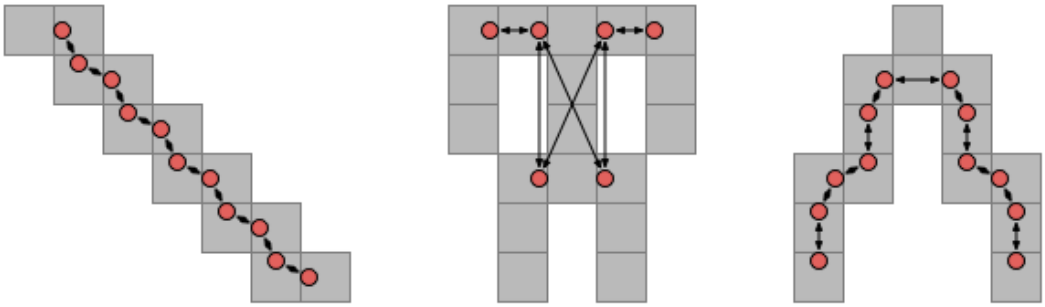

Figure 10.2 : *Placement of the oscillators and the links between them. Left : the snake. Middle : the tetrapode. Right : the "arrow".*

10.2 Genetic Algorithm

The GA is the same as for evolving the oscillator and the value of the different parameters are the same too. The pruning operator is not yet useful and is thus removed.

10.3 Encoding and fitness function

A chromosome encodes the left and right tonic drive (BS), the offset of each oscillator and the extent of the projections of the connections to corresponding neurons in neighboring oscillators for each oscillator. We understand by "offset" the position around which the motor oscillates. A standalone oscillator produces an oscillation centered into zero. Although an oscillator can have a different offset when it is coupled with others, some offset's values may be unreachable. Thus, some offsets can be very efficient for locomotion of the robot but impossible to reach. Therefore, a gene is present for each oscillator and its value is added to the oscillator as an additional offset. The range of the projections of the connections is measured in terms of the number of oscillators. As the oscillator used has 8 connections (not including the connections from the brainstem and a left-right symmetry assumed), and as the projections are the same in all directions, we need 8 genes for coding the extent of the projections for each oscillator. The length of a chromosome is thus $length = 2 + (1 + 8) * nb\_osc$ with nb_osc the number of oscillators in the robot.

The parameter values are directly encoded with the following boundary values: 0.2 and 1.12 for the left and right BS, $-\pi$ and π for the offset and 0 and a variable value for the range of the projections. The upper limit of the projection is 3 for the tetrapode, 4 for the arrow and 5 or 6 for the snake depending of the run. The weight of the connections between oscillators correspond to those of the connection into the oscillator.

The individuals are tested during 30 simulated seconds. Throughout this period, the robot must cover as long a distance as possible. Several fitness functions have been used that measure this distance, mixing straight distance and cumulated distance and favouring or not a direction. Indeed, the simple measure of the straight distance from the initial location to the final location is not sufficient because of the risk to return to starting point after 30 seconds (if the robot makes a circle for example). Moreover, knowing the shape of the modular robot we can be inclined to think that a specific direction is the right one and thus favour it.

The first fitness function is based on the covered distance (arrival position minus starting position) plus the cumulated distance (each second the position is recorded so we can evaluate precisely the real covered distance) (Table 10.1). The next three favour one direction over the two others by using only the distance in the favoured direction instead of the absolute distance and by substracting the distance in the two other directions. Finally, the fourth favours the horizontal plane and does not use anymore the cumulated distance.

N°	Type	Formula
1	generic	$f = \alpha \cdot \|\vec{p_N} - \vec{p_1}\| + \beta \cdot integrated\_traj$
2		$f = \alpha \cdot x^2 - \beta \cdot integrated\_traj - y^2 - z^2$
3	favour one direction	$f = \alpha \cdot y^2 - \beta \cdot integrated\_traj - x^2 - z^2$
4		$f = \alpha \cdot z^2 - \beta \cdot integrated\_traj - x^2 - y^2$
5	favour XY plane	$f = \sqrt{x^2 + y^2} - \|z\|$

Table 10.1 : *The different fitness functions used. f represents the measured performance, where $integrated\_traj = \sum_{i=1}^{N-1} \|\vec{p_{i+1}} - \vec{p_i}\|$, p_i is the i^{th} point sampled on the trajectory of the robot, N is the total number of recorded point, x, y and z are the distance projected onto the correspondent axis. and α and β are coefficients that modulates the weights of the absolute and integrated distances. Here : $\alpha = 1$ and $\beta = 0.3$ in order to favour the robots moving in a straight way.*

10.4 Results

The snake Three sets of runs of respectively three, four and three runs each were done. The two first sets don't have the parameter specifying the offset into their chromosome. The first set uses the generic fitness function and the results are bad. In Chapter 8 we used the same structure with a simple sinusoidal controller and we get a good locomotion in the "natural" direction. But this is not the case here. The snakes bend in two and move, changing direction very often.

The second set of runs uses two times the fitness function n°1 and two times the n°2 (Table 10.1). The locomotions found by fitness function n°1 - which favour the X direction - are efficient but again the two individuals opt for a "bend in two" technic. The trajectory is a spiral (Fig. 10.3) but this is not a problem because the center of this spiral is the X axis. Thus we can consider that the robot goes straight.

As the two first sets of runs didn't find the same locomotion technic as those found during first tests, the offset parameter was added to the genome of the robots. As previously discussed, when the offset is only a consequence of the coupling of the oscillators, some offset's values are unreachable. Thus, the cause of the problem could have been that the necessary values are not reachable by the oscillators. This is not the case, but the performances of the snakes of this third generation are better than the previous. Three runs were carried with the fitness function n°0, n°1 and n°4. All converged to good behaviour but particularly those with fitness function which favour the X axis.

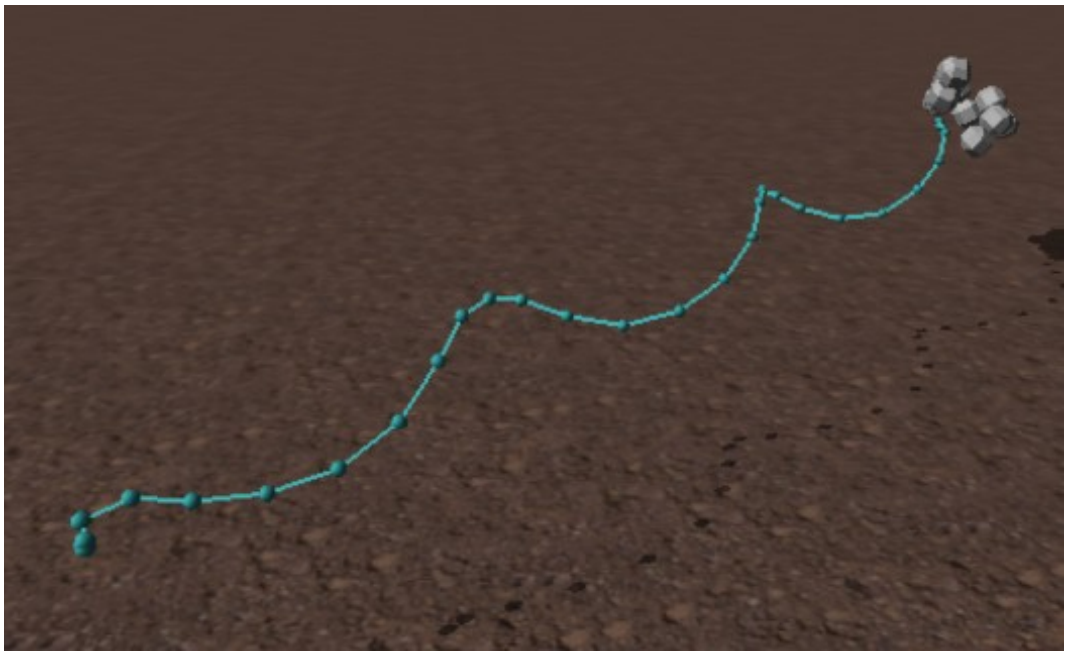

Figure 10.3 : *Snake and its spiraly trajectory (in blue).*

The tetrapode Three runs only were carried out for 1000 generations with this structure because of the very simple behaviour necessary for moving. All the runs converged to robots capable to move (Fig. 10.4) with a very similar technic. Two runs have the generic fitness function (they converged to a very similar genom) and the third have the fitness function which favour the horizontal plane.

The velocity is not high but this imperfection is certainly due to the basic

shape. On the other hand, the trajectory is not perfectly rectilinear though, it seems not to be too difficult to find a synchronisation between the four limbs that permits that. I think that it is the consequence of the placement of the oscillator and the connection between them which is not optimal.

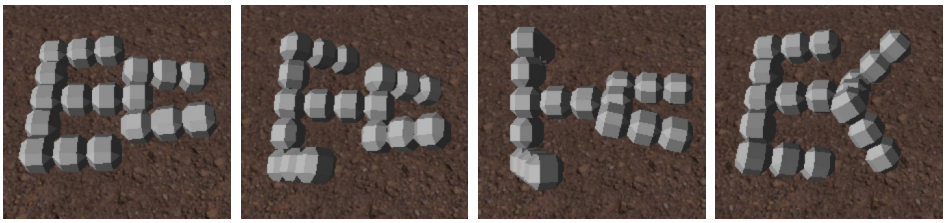

Figure 10.4 : *Locomotion of the tetrapode (the robot is going to the left). For a more explicit view of the locomotion see the movie on [36]*

The arrow Four runs were carried with the generic fitness function. Two of them lead to a good locomotion, in the "natural" direction expected (Fig. 10.5). The robot uses its two limbs as fins and its trajectory is rectilinear. The motoneurons activity reveals a synchronisation of the twelve oscillators (Fig. 10.6). The two limbs have the same dephasing but this results in an anti-phase behaviour because they are mirrored. Each limb is separated into two blocks that have a time difference.

Figure 10.5 : *Locomotion of the "arrow" (the robot is going to the right). For a more explicit view of the locomotion see the movie on [36]*

One other of the four runs attempts to swim in the contrary direction by tucking in its limb. The result is poor, but this must be a strong attractive local optimum because the GA was unable to find another possibility once this was found. Finally the fourth uses the ground by going down. Once it's on the ground, it leans against it for moving and rotates for going again on the ground. This locomotion is efficient but less than the first described.

One can be surprised by the placement of some oscillators on a joint that appears to be useless. But if we remove the useless oscillators (Fig. 10.7) the result is pathetic. Maybe this result is due to the too little number of remaining oscillators that are unable to create the complex behaviour desired.

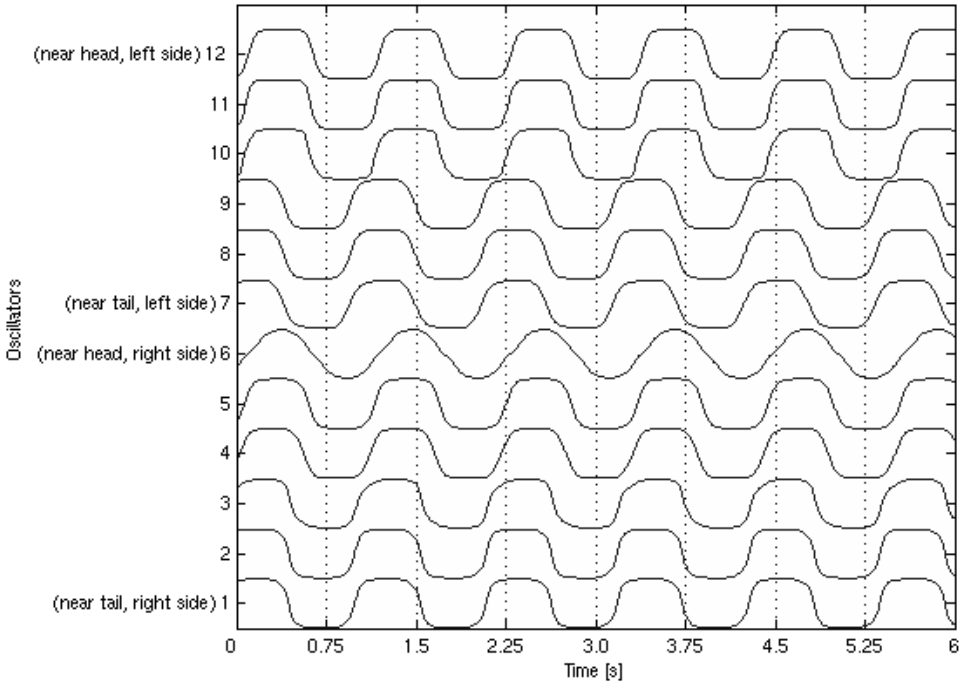

Figure 10.6 : *Neural activity of in the 12 oscillators. Each curve corresponds to the difference between left and right motoneuron. The six upper are the left limb, the 6 others the right.*

Figure 10.7 : *Left : Placement of 8 oscillators instead of 12 into the arrow robot. Right : the spiral structure.*

10.4.1 Discussion

Firstly, we had imagined at the beginning a fourth form: a spiral (fig. 10.7). It seemed to us that this structure corresponded well to the nature of the axes of rotation of Neubot modules and to the aquatic environment. However the tests carried out with this form showed that GA brought back this form to a snake by removing the offset put manually to obtain the spiral. Moreover, if

we block the offsets to keep the form by only authorizing an oscillation around the starting position, the results become very bad as well with simple sines as with the oscillators coupled by using the synaptic spreading. So we gave up this structure.

The way in which we implemented the synaptic spreading differs from that of A. Ijspeert [3]. First of all, the distances of projections are symmetrical on our premises, i.e. an oscillator will project as much towards the oscillator of the father's module as that towards its sons. That induces a smaller genome but also less flexibility. Perhaps some solutions were ignored because of this choice but the limitations of time encouraged us to limit the number of free parameters. Furthermore, A. Ijspeert did a rescaling of the weight of the connections from other oscillators. This is not our case. We test without rescaling and no saturation appears. Indeed, as the BS's are positive and the neurons of type A and C are negative, the inputs seem to compensate together.

As effective locomotions could be found for each three chosen structures, the synaptic spreading seems to be an interesting option for the coupling of the oscillators. However, it is necessary to keep in mind some problems like the incapacity of the GA to find the locomotion of the snake realized with simple sine. Moreover, although it advances, the tetrapode is by no means fast. Much more than the synaptic spreading, it is the form which is reassessed. That shows the difficulty of imagining optimal forms for our modules and much more so in a water environment.

Chapter 11

Co-evolution

In nature, the controller (the neural system) and the structure (the body) are evolved in a parallel way. A change of one leads to an adaptation of the second which leads to a coherent and adapted to its environment life system. Up to now we evolved only the controller, firstly evolving the oscillator based on a neural network and then evolving connectivity between these oscillators in a modular robot predefined in order to lead to an efficient locomotion. The following stage consists of not only evolving the brain but also the morphology of the robot. Moreover, while allowing it to be closer to a natural way (thougth this is not an aim in itself), it makes it possible to avoir pejoring the locomotion of the robots by unsuited morphologies due to our difficulty of imagining new and different forms from what we see in reality. Indeed, the particular axis of rotation of the modules (parallel to the normal of the faces) is not represented in nature and we are inclined to use forms adapted to another type of movements based on the use of regular hinge joints which are not well adapted for our needs.

The concepts used in this chapter as well as certain details of implementation are inspired of the work of Daniel Marbach [29].

11.1 Encoding

The genotype is a tree (no cycle allowed) as in [6], each node representing a module (Fig. 11.1) thus it is a direct encoding which strongly correlates the phenotype and the genotypes. The simulation and evolution environment uses the genotype as the internal representation of the robot. In more of the tree, some general informations are encoded in a chromosome of real numbers which we will call *High Level Parameters* (HLP). It contains the left and right input drive (BS) and the different probabilities of the GA. Indeed, as I will explain later, the probabilities can be coded and evolved with each individual.

A node contains a chromosome of 14 genes. The faces of a module are numbered as follow : 0 for the face in direction of the X axis in the local coordinates

system, 1 for the opposite, 2 for the Y direction, 3 for the opposite and finally 4 for the Z direction and 5 for the opposite (Fig. 11.2). A module is always attached to its father on its face zero and therefore can have five children at most that can be put on the faces numbered 1 to 5. This is encoded with five integer, each number between 1 (included) and 5 (included) appearing once (5 genes).

The nodes contain moreover the angular offset of the joint between it and its father (1 gene). Finally, as in the previous chapter, each internal connection of the oscillator needs an integer specifying its range for the synaptic spreading. As each module is connected to its father by only one joint placed on the face zero of the child and as no cycles are allowed, each module has at most one motor actuated (always on the face zero) and thus only one oscillator per module is needed. Therefore, only 8 additional genes are added to the chromosome.

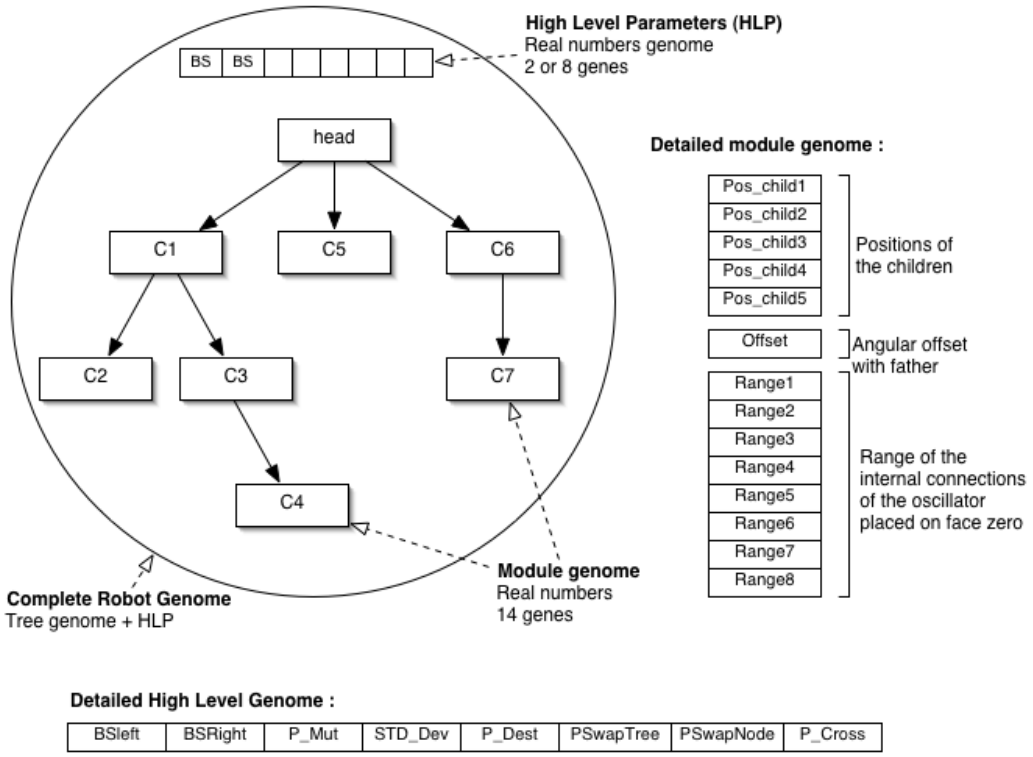

Figure 11.1 : *Genome of a robot.*

11.2 Genetic algorithm and fitness function

At the beginning of the GA the population is initialized with N randomly created robots. Then, the GA uses mutation and crossover as genetic operators in order to create C new robots each generation (C robots will be removed after selection). The mutation acts at different levels (Table 11.1) : on the tree, on the node's

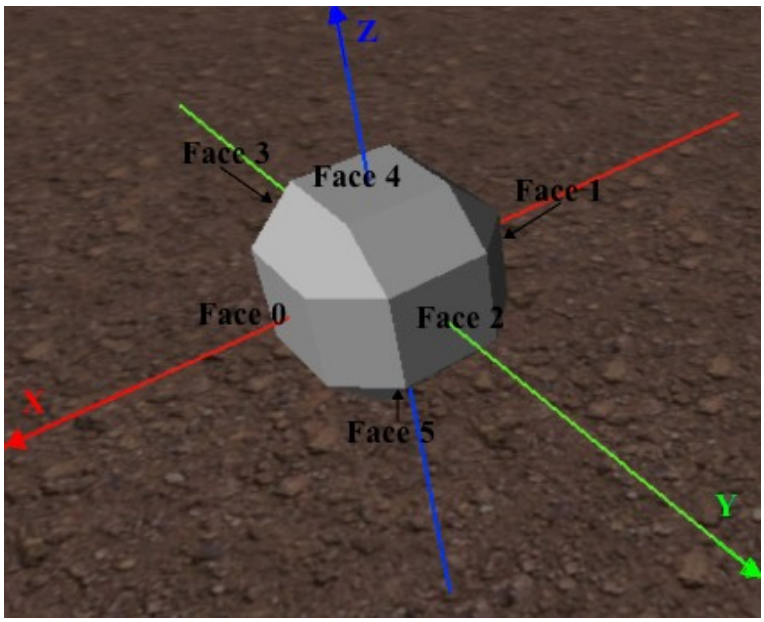

Figure 11.2 : *The numerotation of the faces of the modules.*

chromosome and on the High Level Parameters (HLP). Different mutations can occurs on the tree, that can destruct or produce a node, or swap two nodes or subtrees. For the nodes and the HLP, if mutation occur on a numerical parameter, a random value from a gaussian distribution is added. For the five integer genes specifying the position of the children, we test only once the mutation and if it occurs, then we permute two values of the five.

Thanks to the tree structure of the genotype, the implementation of the crossover is not difficult. It can produce one or two children. A child is formed by copying the mother or the father and replacing one of its sub trees with a sub tree of the other parent.

For selection we used the rank-proportional roulette wheel method proposed by GAlib. This selection method picks an individual based on the magnitude of the fitness score relative to the rest of the population. The higher the score, the more likely an individual will be selected. Any individual has a probability p of being chosen where p is equal to the fitness of the individual divided by the sum of the fitnesses of each individual in the population.

Two ways exist to set the different mutation and crossover probabilities. The first allows the user to explicitly give global values that are the same for all individuals (the values used in this chapter are displayed in Table 11.2). On the other hand, the probabilities can be different for each robot and evolved with it. This way is directly taken from the Adam project of Daniel Marbach and only some tests has been made here with it.

The fitness function used in this stage is the same as in chapter 10.3 and

numbered '1' in Table 10.1. We don't use the functions that favour a direction because we don't know in advance the shape of robots and therefore the direction to favour. A robot begins to be evaluated after *start_time* seconds and finishes at *start_time* + 30 seconds. This duration is neither too long (if it was, the GA would be very slow) nor too short (if it is, the results are biased because a simple jump at the beginning of simulation is sufficient to get a good evaluation).

Type			**Description**
Mutation	Tree	destructive	A node randomly picked (probability P_{struct}) in the tree is deleted with its subtree if exist.
		productive	At every node that doesn't have the max children, a new module is attached with probability P_{struct}.
		swap node	This rearranging mutation operator randomly picks 2 nodes in the tree and swaps them.Any node has a P_{struct} chance of getting swapped, and the swap could happen to any other node.
		swap subtree	This rearranging mutation operator does the same thing as *swap node* but swaps the nodes as well as its subtrees that are selected (probability P_{struct}).
	Nodes	offset & ranges	A random value from a gaussian distribution is added to the real value mutated with probability P_{mut}.
		children's pos.	Swap two positions of the array of children's positions with probability P_{mut}.
	HLP	all	A random value from a gaussian distribution is added to the real value mutated with probability P_{mut}.
Crossover	Tree		Generate one or two new robots by taking copying one parent and replacing one of its sub trees with a sub tree of the other parent (appears with probability P_{cross}).

Table 11.1 : *Summary of the different genetic operators used in the GA. First the mutation operator on the three level of the genome, then the crossover (only on the tree).*

	N	C	P_{struct}	P_{mut}	P_{cross}
Value	100	50	0.025	0.6	0.5

Table 11.2 : *Parameters of the genetic algorithm.*

11.3 Implementation

This section will briefly explain the implementation of the previously decribed genetic algorithm. A partial design diagram is shown in figure 11.4 whose simulator's and CPG's classes are not all displayed. A *Robot* notion has been added which contains at least one module (the head) and which simplifies the decoding of the genome. Indeed, the whole genome is decoded at robot level and not partial genome in each module. The *Simulation* has been adapted to that and becomes *RobotSimulation*. Only one robot can be placed in the simulation at the same time and when it is placed a verification of the robot is done. If the robot does not yet have a module because of a mutation the placement fails. Moreover, a robot can have a structure which stacks several modules at the same location. In order to detect such a problem, I use the collision detection before the beginning of the real simulation and if collision exists, the placement of the robot in the simulation fails and a fitness value of zero is given it.

The GeneticAlgorithm class contains the initialization of the GA, and the fitness function in which the robots are created and placed into the Simulation during 35 seconds (5 seconds of stabilisation and 30 seconds of evaluation). The tree genome is implemented in the Genome class which is a sub-class of the GATreeGenome from GAlib. The genetic operators that are not supplied by this library are in GeneticOperators.

It is not sufficient to only create oscillators for all the modules we must also connect them together. We use the following rules (see figure 11.3 for an example):

1. If possible always connect an oscillator with the oscillator of the father's module.

2. As the head has no oscillator, connect the oscillators of its children together

If the modules can have no oscillator even they are not the head (last set of runs, see below):

1. If possible always connect an oscillator with the oscillator of the father's module or if the father doesn't have an oscillator recursively try to find an oscillator in the grandfather and so on.

2. As the head has no oscillator, connect the oscillators of its children together. If a children of the head doesn't have oscillator, recursively try to find the oscillators of the grandchildren.

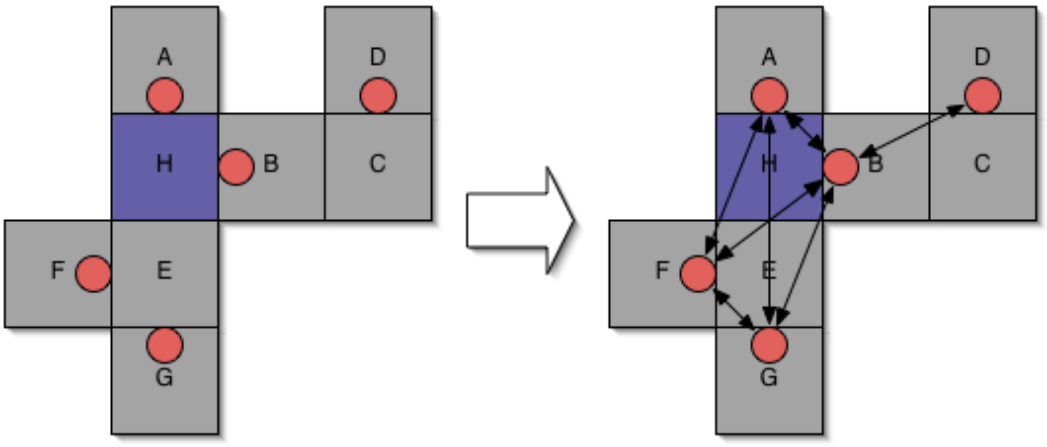

Figure 11.3 : *Example of the connections of the oscillators. Each square represents a module, the named H is the head. The red circle are the oscillators. The modules don't have all oscillators. Firstly the module D tries to connect to its father C but C doesn't have oscillator. Thus, recursively, it connects to B. Then around the head, we collect all the modules. A and B are easy, but E doesn't have oscillator. Recursively we search oscillator in its children. Therefore we find four oscillators in A, B, F, G and we connect them together.*

The simulator has been configured in order to be not too accurate, but very stable. The values of all the variables can be found in Table A.4. Despite that, some unstabilities can appear in some precise cases. It is the case if more than three modules are aligned. This problem is a known problem of ODE with rotating bodies [34]. To avoid this, a test is done each second to check the angular velocities of all modules in simulation and a null score is given the robots which have modules with excessive angular velocity. Unfortunately, that removes robots which would be very efficient in reality. Perhaps the next version of ODE will correct this problem.

Another problem (one could say feature!) of ODE was the cause of one of the biggest problem I met during the implementation. When a variable of an ODE's simulation is set to *nan* (i.e. Not a Number), all the variables of the simulation are set to *nan*. Therefore, it is very difficult to find what is the cause of the problem, above all if we don't know that ODE does like that.

Debugging and optimisations A certain time was devoted to the debuggind and above all to the optimisations. As the genetic algorithm creates hundreds of thousand of robots and simulates each for 35 seconds, the least memory leak is disastrous and bring the program to the crash. I used thus LeakTracer and Valgrind for detecting and resolving these memory leaks. As the program is relatively big and uses a lot of external libraries, some very little leaks was not found. Although all the problematic ones have been eradicated and the GA can turns for a week without crashing, I developed a little program whose aim is only

to launch the GA and to restart it if it crashes retrieving the same population, seed, etc...

The profiling is also very important. A little optimisation of a method where we are 15% of the time results in a spectacular time saving. Therefore, the *LeakyIntegrator* and *CPG* classes developed previously are very tricky. As the oscillator chosen has only six neurons, a new *CPG* class was coded taking that into account. Moreover, all the calculations from the *LeakyIntegrator* class are optimised. Finally, the duration of a set of simulation test was divided by twelve!

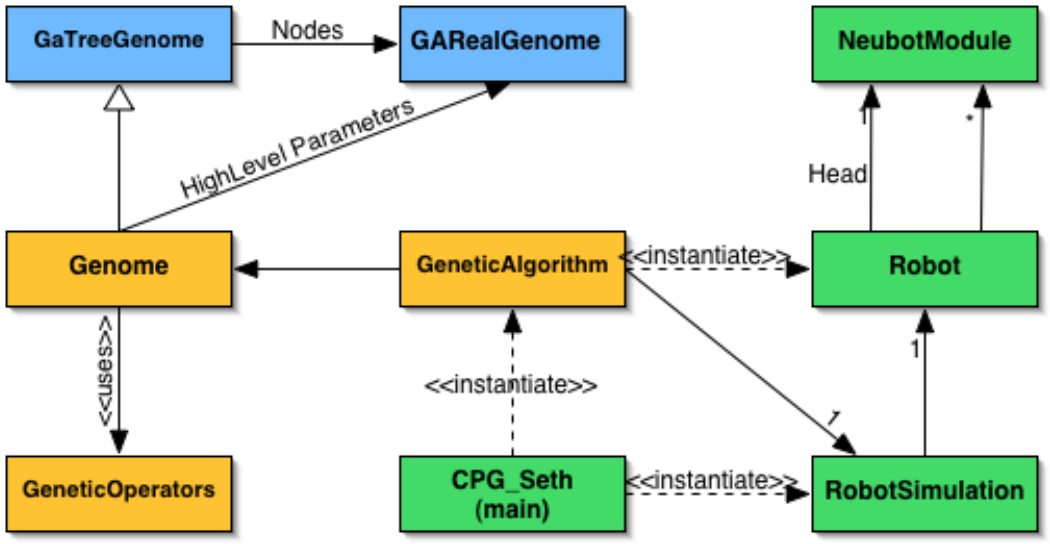

Figure 11.4 : *The design diagram of the classes developed for the coevolution. Green boxes: simulator's classes. Orange boxes: GA's classes. Blue boxes: GAlib's classes.*

11.4 Results

Four sets of eight runs each were carried out for 1000 to 2000 generations. Obviously dozens of runs were carried out with the purpose of testing the simulator and the GA, but this chapter describe four precise sets of runs whose parameters have been precisely chosen and whose number of generations was big. Only the most efficient robots found are described here but all the initial conditions of the sets are detailed.

First set The first set uses the generic fitness function and six of the eight runs converged to robots able to move quickly and without turning - but sometimes with spiral trajectory. The projection's distance of the inter-connections of the oscillators can be varied from zero to four (measured in terms of the number of oscillators). The maximum number of modules was set to 15 for all runs and the

initial number (i.e. the maximum number of modules for the first generation) was different according to the run. Three of them began with five modules, two with three, one with eight, one with seven and one with ten (the results of the two last are bad).

Those with three initial modules converged to the same robot with four modules (Fig. 11.5 left). It is very simple but efficient with a velocity of 9.5 [m/min]. The friction maybe is a problem in reality with such robots but it seems to synchronize the moving modules avoiding this problem.

The robots with five initial modules converge to two different robots, one with five modules (two robots by three), the other with six. The first (Fig. 11.5 middle) is very efficient with approximatively a velocity of 11 [m/min]. Its trajectory is almost rectilinear but directed towards the top and the robot rotates during its locomotion. The four oscillators are totally synchronized (Fig. 11.6) and symetric. As the two sides of the robot are mirrored, the synchronisation of the oscillators results in an anti-phase between the two sides (Fig. 11.7). The second robot, with six modules, has less good results than the previously described robot (velocity of 9 [m/min]). Its shape is shown on figure 11.5 on right.

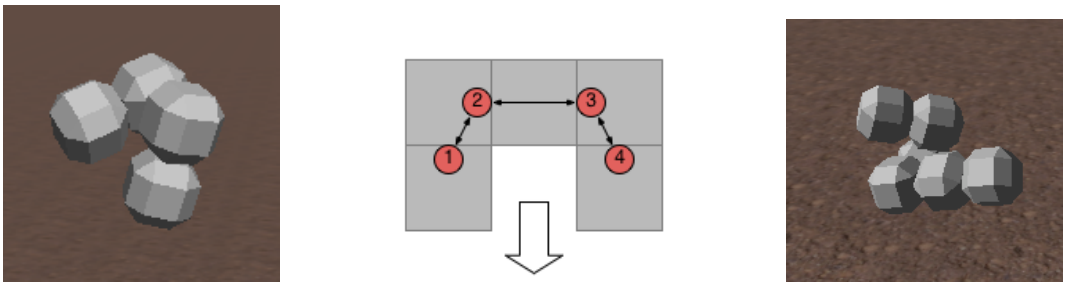

Figure 11.5 : *Left: The 4 modules robot. Middle: The 5 modules robot with the numerotation of the oscillators. The arrow specifies the direction of the locomotion. Right: The 6 modules robot.*

Second set Eight runs were done with the fitness function favouring the horizontal plane. The initial conditions, except the maximal initial number of modules, was the same as in the first set of runs. To favour solutions with more than 4 modules none of the runs had less than 5 modules as an initial number. Three of them had 5 modules, and the other respectively 6, 7, 8, 10 and 15 as initial number of modules. All except one converge to robots able to move in a direction with a rectilinear or spiral trajectory. Several of them found again robots of the first set, thus we don't describe them in detail.

Three runs (initial number of modules : 5, 5 and 6) converge to the robot with 5 modules already found in the first set (Fig. 11.5 middle, 11.6 and 11.7). One converges to the four modules structure shown on Figure 11.5 on left and

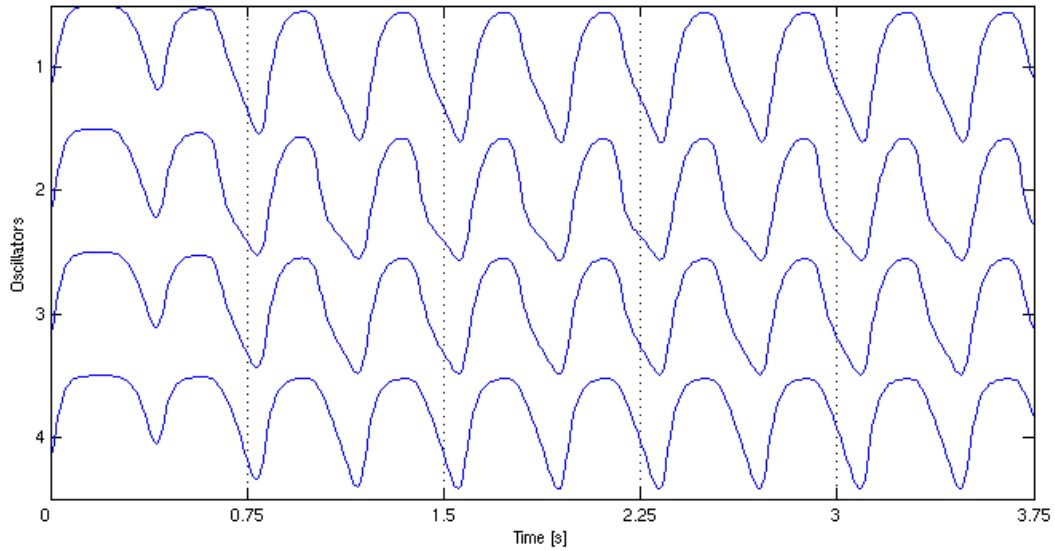

Figure 11.6 : *Neural activity of the robot with five modules (therefore four oscillators) evolved by coevolution. Each curve corresponds to the difference between left and right motoneuron of one oscillator. The numerotation of the curves corresponds to the numerotation on Figure 11.5 (middle).*

Figure 11.7 : *The locomotion of the robot with 5 modules (direction: left). The offset of the angles can be observed.*

two to a variant with an additional module at the extremity of a limb (Fig. 11.8 left). It is the case for the run with 15 initial modules per robot. This structure is unstable if the friction coefficient is increased and may not be usable in reality. Finally, the run with 10 initial modules converge to a robot with 8 modules (11.8 middle) able to move with a velocity of 7 [m/min]. This robot does not seem to be optimal because of the module indicated by the red arrow on the figure. But if we remove it the robot becomes unstable because of the five aligned modules. Thus this structure is optimal with this simulator and its flaws but certainly not in reality.

Third set As the two first sets of runs shows that the robots have difficulty in adding or removing modules (see the discussion in section 11.5) a new set was

Figure 11.8 : *Left: A variant of the four modules robot but with one additional module amplifying the oscillations. Middle: A eight modules robot. The red arrow shows a module which is useless for locomotion but necessary for avoiding unstabilities. Right: Same as middle but with one additional module.*

launched with a fundamental difference: when a module is added all the range genes are set to zero. Therefore, a new module doesn't perturb the preexistent neural network. Of course the ranges are evolved normally then. I hoped that this change would encourage the GA to add modules to the robots but this was not the case.

Although the change doesn't improve the addition of modules, the results are as good as the other set of runs. The GA found again the same robots. Amongst others, a run with 10 initials modules converged to the solution illustrated by the figure 11.8 (right) which is similar to the one of the second set but with one additional module. This third set gives as well a very good robot. It is a variant of the robot with five symetric modules but with a "tail" between the two limbs (Fig. 11.9). The limbs roll on the tail avoiding the frictions and the tail prevents the robot from rotating too much. Its trajectory is very rectilinear and its velocity is near 11 [m/min]. The robot goes in the contrary direction as the robot with 5 modules, what is more natural.

The oscillators activity is displayed on figure 11.11. The oscillators are totally synchronized. The two short limbs have same oscillations (oscillators 1-2 and 5-6) and the central limb oscillates too although this seems useless a priori. But these oscillations on one hand influence and contribute to the other oscillations and on the other hand make easier the rubbing with the two short limbs.

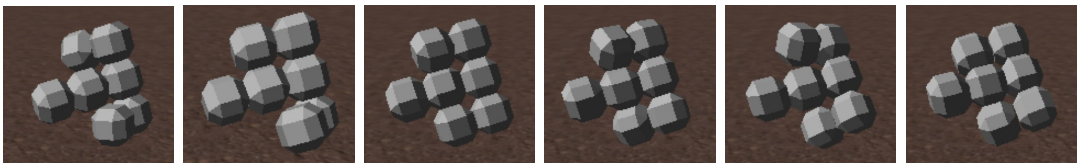

Figure 11.9 : *The robot with 7 modules. It moves to the right.*

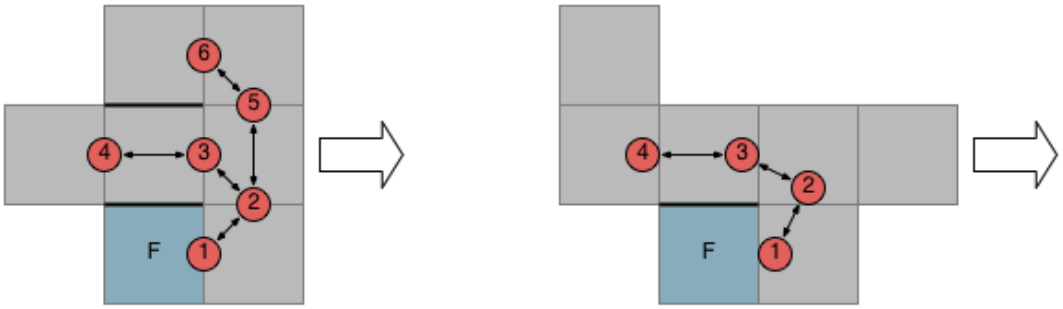

Figure 11.10 : *Placement of the oscillators and their links. The father module (the root of the tree genome) is blue with a 'F'. Left: the robot with seven modules evolved during third set of runs. Right: the robot evolved during the fourth set of runs.*

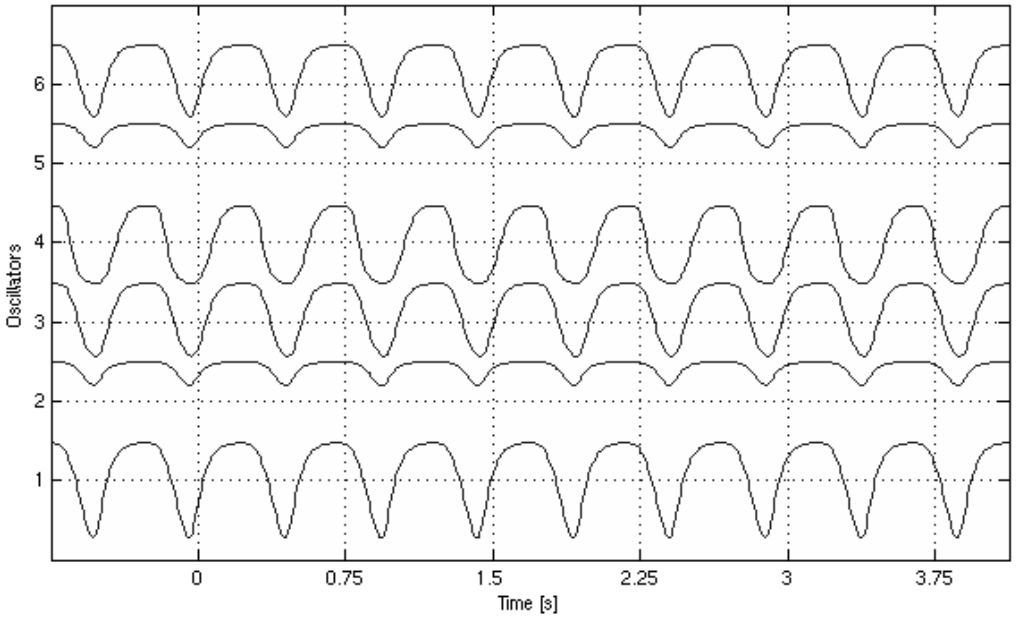

Figure 11.11 : *Neural activity of the robot with seven modules (therefore six oscillators) evolved by coevolution. Each curve corresponds to the difference between left and right motoneuron of one oscillator. The numerotation of the oscillators corresponds to the numerotation on Figure 11.10 left.*

Fourth set A fourth and last set was launched with a major change. Until now all the modules have an oscillator which can be quiet depending of the coupling with the others. In this set of runs, the modules can be *solid*, i.e. there is no oscillator and no motors actuated. Therefore, a module doesn't depend yet on the coupling between oscillators to be *solid* or not. An additional gene is added to the genome of the modules (the nodes of the tree) which indicate whether the

module is actuated or not. The coupling between the oscillators changes a little bit due to that as explained previously in chapter 11.3 page 57.

Eight runs were carried out for 1000 generations with 100 individuals. They have different *INM* between 3 and 10. All runs, except one, converged to robots able to move more or less fast without turning. Two runs converged to the same robot with five modules as which is showed on figure 11.8 on left. Two others leads to the robot with six modules displayed on figure 11.12. It achieves a velocity of 10.3 [m/min] with a very straight trajectory. Finally, a new structure appears which use the *solid* modules. This robot is made of seven modules (Fig. 11.13), but only four joints are actuated (Fig. 11.10 right shows where are placed the oscillators). Its velocity is 10 [m/min] and the trajectory is good. The neural activity is shown on figure 11.14. The four oscillators are synchronised without dephasing. We can observe the diversity of the behaviour of the neural network generating the oscillations. The oscillator 1 is near a sine what is not the case of the three others. The numbered 4 is closer a triangular signal.

Figure 11.12 : *The robot with six modules evolved during a run of the fourth set.*

Figure 11.13 : *The robot with seven modules evolved during a run of the fourth set.For a more explicit view of the locomotion see the movie on [36]*

11.5 Discussion

About the implementation's choices Firstly, the tree structure of the genome prohibits having cycles in the robot. That can appear to be a limitation, but with the type of joint used with the Neubot modules, it is not envisageable to have cycles. Then, an important point for the structure of the robots generated by GA is the way in which the tree is built. It can either be in-depth, or in width.

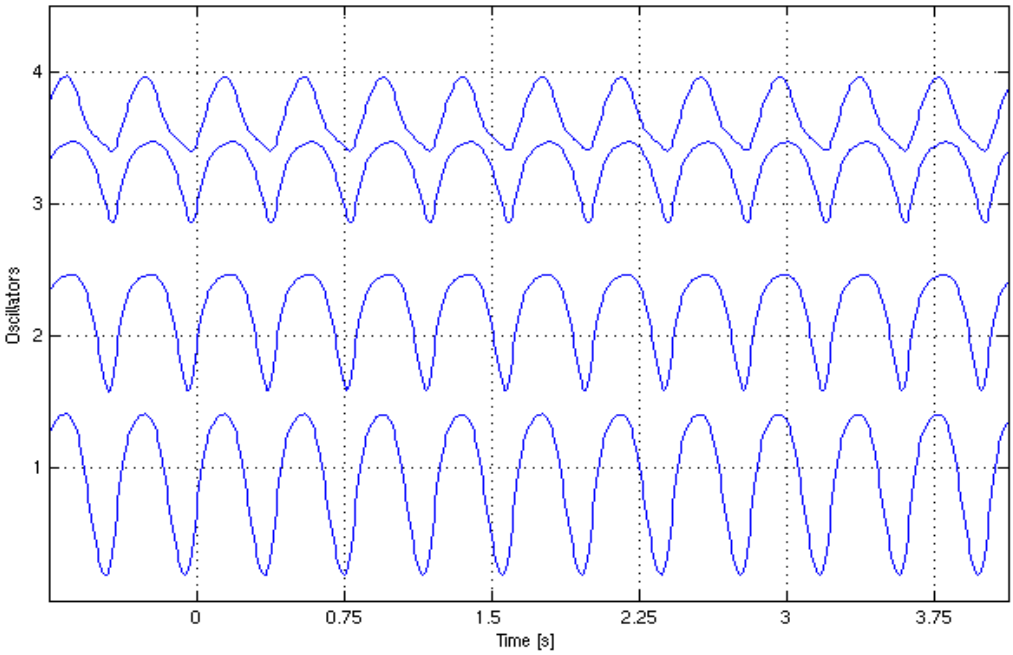

Figure 11.14 : *Neural activity of the robot with seven modules (but only four oscillators) produced by a run of the fourth set. Each curve corresponds to the difference between left and right motoneuron of one oscillator. The numerotation of the oscillators corresponds to the numerotation on Figure 11.10 on right.*

Being given that the number of modules is limited, in-depth construction will support robots with long and not many members whereas construction in width will encourage "very bulky" robots with a lot of short members. In described simulations, we used in-depth construction because that decreases the risks to stack modules at the same location. However the other way of constructing also has advantages and it would be very interesting to compare attentively and in detail the influence of each of the two types of construction.

Another important choice is to make the robots symmetrical or not. In nature, the majority of the animals have an axis of symmetry. For an efficient and controlable locomotion, that seems logical. However, I made the choice not to force this symmetry and to see if it appeared spontaneously. That was the case for some robots, but by far not for all. Those which are not symmetrical have a trajectory in spiral what enables them to go right on a long distance but not locally. Another problem appears when symmetry is not forced. The robots which have a symmetry then tend to remain very small and not to add modules to interesting structures. Indeed, if they add a module only to one of the limbs, the robot will be assymetric and will move more badly than before and will thus be eliminated. It is necessary thus that two changes take place at the same time so that a module is added on the two symmetrical members. That undoubtedly

explains why the symmetrical robots with 5 or 7 modules did not grow which would have enabled them to move more quickly with longer limbs.

About the results The results show that simple forms are completely able to be driven in a fast way and without turning. However some points seem to pose problems. The problem of symmetry was already mentioned and we will not speak about it again. The first two set of runs show that the robots have a lot of trouble in adding or removing modules and often confine themselves to the number of starting modules. That is partially explained by the fact that all modules have a neural network which is connected to the others. If a module is removed, the whole network may be strongly disturbed. In the same way, the addition of a module can influence largely the global network (an example is shown on figure 11.15). This is why a third set of runs was launched with a different policy for the addition of modules. A module, when it is added, has all its genes specifying the distance of projection of its internal inter-connection put at zero. Then they could be evolved for gradually integrating the new oscillator into the entire neural network of the robot. Unfortunately the results showed that the influence of this change was poor. Moreover, this change doesn't solve the problem of the withdrawal of a module.

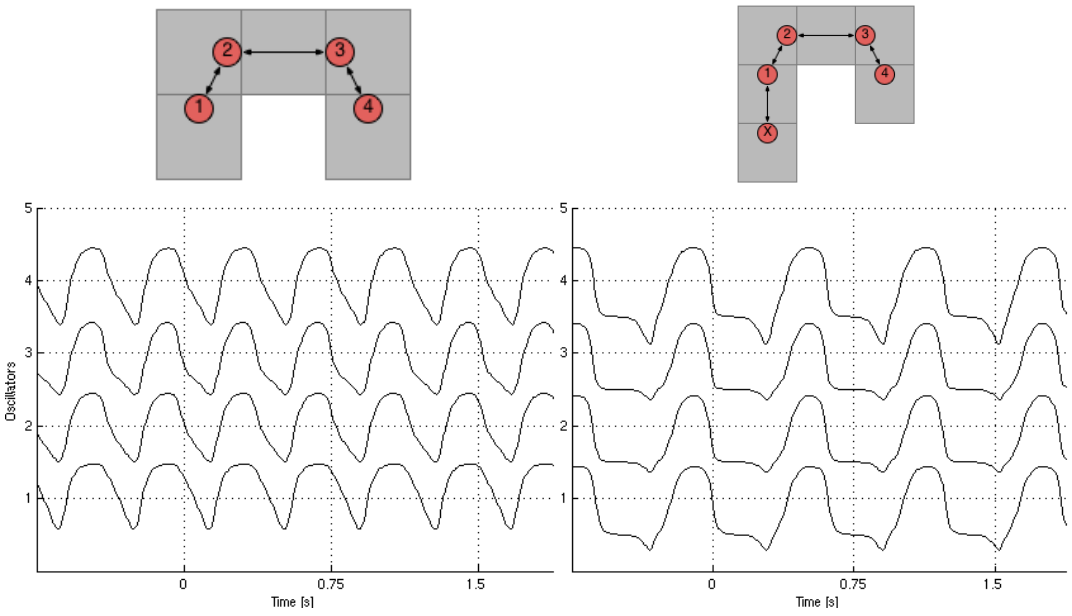

Figure 11.15 : *Adding only one module perturb the general neural network. Left: an already studied robot. Right: the same one with one additional modules whose range's genes are randomly initialized. The behaviour is strongly modified.*

The fourth set of runs add the possibility of having not actuated modules. That could already be the case before according to the coupling of the oscillators

bringing one of them to saturation. But, with according to this way, optimal combinations of modules for the locomotions (for example, to influence the father module but to have a non-actuated son) was quite simply not realizable. It is the same problem which had led us to dedicate a gene to the offset instead of using the offset generated by the network of neuron. This last group of runs did not generate exceptional individuals but on the contrary did create again robots already seen before. It is interesting to note that the majority of the runs thus converged towards solutions not including *solid* modules . The robot that we described in details in the chapter *Results* uses them and its results are good but not better than those of the robots found previously.

What's more, the local optima seem to be very attractive and prevent also the robots from growing or shrinking. The maximum number of modules authorized when the genome is constructed (henceforth called *INM*, i.e. Initial Number of Modules) thus has a great importance. The results show that the runs launched with the same *INM* tend to converge towards the same solution. However, beyond a certain *INM* (approximately 10 modules), the robots are so bad and there are so many which are eliminated because of modules stacked on same location, that the runs will converge to robots having a significantly low number of modules. This convergence is not any more correlated with the *INM* but depends on the robots of the starting population which have less modules and is thus random.

Concerning the genomes, those of the robots found by the GA are very different and no overall characteristic can be observed. Moreover, a χ^2 test confirms that the range's genes are randomly distributed, except for the zero value which appears more once on two.

11.6 Comparison with another controller (continuous rotation)

The motors of the Neubot module were used up to now in an oscillatory way. However, they could entirely being used in a continuous way. If we did not use this possibility up to now, it is because the majority of study are interested primarily in the locomotion based on oscillators and because nature gives few, or no, examples of continuous rotation. To study the possibilities offered by continuous rotations, we developed a very simple controller which makes the motors turn at a given angular velocity.

Encoding and GA The encoding remained the same one for the tree and the High Level Parameters. On the other hand the nodes (i.e. modules) do not have yet the same type of chromosome. The first six genes remain the positions of the children and the offset. Then the genes coding the "ranges" are replaced by a single parameter specifying if the module must be actuated or not. Other possi-

bilities were offered to us like the possibility of coding the angular velocity of the engine or an angular dephasing with the father. The first possibility was rejected because that prevented from keeping a fixed shift between the actuated modules and the second would have been redundant since the gene 'offset' specifies already the angular shift with the father. Therefore the motors of all the actuated modules are rotating at the same angular velocity. The genetic algorithm used is strictly the same one as that used up to now.

Results Six runs were carried for 1000 generations each. Two began with an Initial Number of Modules of 5, two with 7, one with 3 and one with 8. All converged to the same neat solution: robot made of 4 modules (see figure 11.16). This robot works like a propeller. The two "limbs" rotate in the same direction, putting in rotation the whole body. As the offset between the two modules of the body is not null, the general shape of the robot is like an helix. Thanks to the shape of the modules, they can rotate despite the fact that they are near each other.

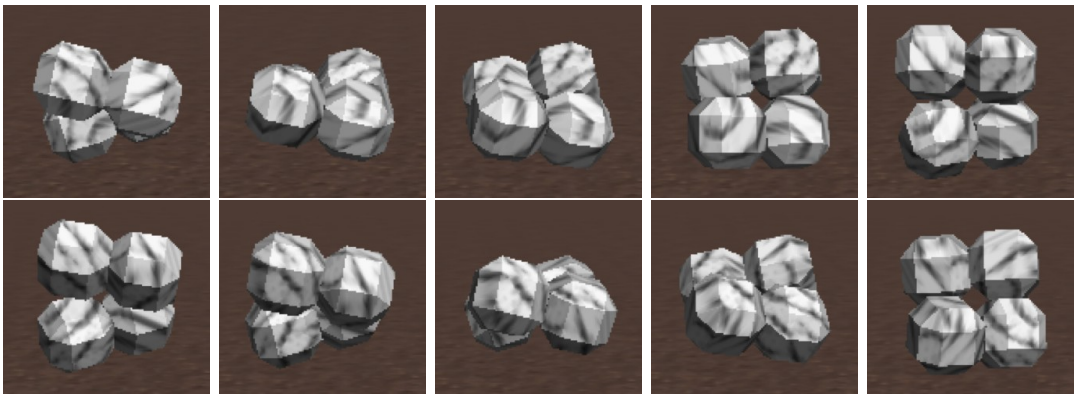

Figure 11.16 : *The unique solution found by the six runs with continuous rotation. Read these pictures as a comic strip.*

Discussion The robot which was found by the six runs must be an optimum extremely large and attractive since no other robot was found. It would be very interesting to test on the first prototypes this type of locomotion to see whether it is realizable in spite of frictions. Moreover, it requires few modules and only one motor for each module, which makes of it an ideal candidate for first tests on prototype.

The best *propeller robot* can achieve a velocity of 9.25 [m/min] which is a little bit lesser than the four modules robot controlled with oscillations (section 11.4 page 60). However, the trajectory of the *propeller robot* is totally rectilinear contrary to the trajectory of the other which is nearly rectilinear but not totally.

The fact that a unique solution was found which is so simple is maybe problematic. This solution is able to modulate its velocity (we tested that changing the angular velocity change the velocity) but not to control its direction. We tried to set a different angular velocity on each limb, but it continued to go straight. In conclusion, this robot is a very good and very simple solution for moving fast and straight with the possibility to modulate the velocity but it is also a dead end because it is not envisageable that we do anything else with it. It would be interesting to know which other solutions the GA can find if we forbid to it to create this robot but due to the time limitation I did not try that.

Chapter 12

Conclusion and future work

This project had two main goals. The first was to carry out a framework for the simulation of modular robots and the second was to carry out a locomotion in an aquatic environment of modular robots. The simulator is functional and was largely used in the second part of the project without major problems. Certain instabilities - partly due to ODE - exist but which can be largely controlled if adequate parameters are used. Moreover, selected software architecture makes easy the implementation of new types of modules or the change of the environment in which the robots evolve.

The underwater locomotion with modular robots was satisfactorily carried out, that is, either with structures defined by us or with shapes defined by coevolution of body and brain. The coevolution showed its might by creating robot's structures of which we had not thought and which prove to be powerful. The techniques used usually with standard joined hinges, like the use of CPGs, proved to be usable with the particular joints of the Neubot module.

The comparison with continuous rotations showed that the two techniques could be usable. Of course, it would be necessary to make other tests with continuous rotation for better understanding of the fitness landscape. Indeed, the fact that all the runs converged to the same solution is surprising. It would also be very interesting to test mixed configurations of continuous oscillations and of rotations in the same robot. Perhaps one of the two options will overcome the other or maybe robots mixing both will appear.

Many ways must still be explored. The evolutionary algorithm showed some limits which appeared by the incapacity of the robots to grow and to leave local optima. The fitness function could be largely improved and particularly in order to have robots which can control their direction and their speed. A first experimentation could be done by a phototaxis technic encouraging the robots to follow a light to favour the emergence of robots able to control their trajectory. Moreover, it would be very interesting to use the sensory feedback to try to avoid obstacles.

Bibliography

[1] A. J. Ijspeert, J. Hallam and D. Willshaw, *Evolving swimming controllers for a simulated lamprey with inspiration from neurobiology*, Adaptive Behavior, Vol. 7:2, pp 151-172, 1999.

[2] A. J. Ijspeert and A. Billard, *Biologically inspired neural controllers for motor control in a quadruped robot*, Proceedings of the 23rd Annual Conference of the Cognitive Soeciety, 2001.

[3] A. J. Ijspeert, *A connectionist central pattern generator for the aquatic and terrestrial gaits of a simulated salamander*, Biological Cybernetics, Vol. 84:5, pp 331-348, 2001.

[4] A. J. Ijspeert and J.M. Cabelguen, *Gait transitions from swimming to walking: investigation of salamander locomotion control using nonlinear oscillators*, AMAM 2003, 2003.

[5] D. Marbach and A. J. Ijspeert, *Co-evolution of Configuration and Control for Homogenous Modular Robots*, In F. Groen et al., editor, Proceedings of the Eighth Conference on Intelligent Autonomous Systems (IAS8), pages 712-719. IOS Press, 2004.

[6] Karl Sims, *Evolving 3D Morphology and Behavior by competition*, Artificial Life IV Proceedings, ed. by R. Brooks & P. Maes, MIT Press, pp 28-39, 1994.

[7] M. Komosinski, S. Ulatowski, *Framestics: Towards a simulation of a nature-like world, creatures and evolution*, ECAL '99, pp. 261-265, 1999

[8] Framsticks website,
http://www.frams.alife.pl

[9] G. S. Hornby, J.B. Pollack, *Body-brain coevolution using l-systems as a generative encoding*, in Genetic and Evolutionary Computation Conference, 2001.

[10] H. Lipson, J.B. Pollack, *Automatic design and manufacture of robotic lifeforms*, in Nature, 406:974-978, 2000.

[11] J. Ventrella *Explorations in the emergence of morphology and locomotion behavior in animated characters.*, In R. Brooks and P. Maes, editors, Proceedings of the Fourth Workshop on Artificial Life, Boston, MA, MIT Press, 1994.

[12] J.C. Bongard, R. Pfeifer, *Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontongey*, in Genetic and Evolutionary Computation Conference, p. 829-836, 2001.

[13] J. Suh, S. Homans, M. Yim, *Telecubes: Mechanical Design of a Module for Self-Reconfigurable Robotics*, proceeding of the 2002 IEEE International Conference on Robotics and Automation (ICRA).

[14] M. Vona, D. Rus, *A physical implementation of the Self-reconfigurable crystalline robot*, in Proc. of the IEEE Int'l Conf. on Robotics and Automation 2000, San Francisco, CA, Apr. 24-28, 2000, p. 1726-1733.

[15] E.H. Østergaard, H.H. Lund, *Evolving Control for modular robotic unit*, in Proceedings of CIRA'03 IEEE International Symposium on Computational Intelligence in Rootics and Automation, Kobe, Japan, p. 886-892, July 16-20, 2003.

[16] A. Kamimura, S. Murata, E Yoshida, H. Kurokawa, K. Tomita and S Kokaji, *Self-Reconfigurable Modular Robot – Experiments on Reconfiguration and Locomotion*, IEEE/RSJ International Conference on Intelligent Robots and Systems, pp 606-612, 2001.

[17] A. Kamimura, H. Kurokawa, E. Yoshida, K. Tomita, S. Murata and S. Kokaji, *Automatic Locomotion Pattern Generation for Modular Robots*, Proceeding 2003 IEEE International Conference on Robotics & Automation, Taipei, Taiwan, September 14-19, 2003.

[18] Modular Transformer website,
http://unit.aist.go.jp/is/dsysd/mtran/English/index.html

[19] PolyBot website,
http://www2.parc.com/spl/projects/modrobots/chain/polybot/index.html

[20] W. Shen, B. Salemi and P. Will, *Hormone Inspired Adaptive Communication and Distributed Control for Self-Reconfigurable Robots*, IEEE Transactions on Robotics and Automation 18(5), pp 1-12, 2002.

[21] W. Shen, P. Will, A. Galstyan and C.-M. Chuong, *Hormone-Inspired Self-Organization and Distributed Control of Robotic Swarms*, Autonomous Robots 17, pp 93-105, 2004.

[22] CONRO website, http://www.isi.edu/robots/conro/

[23] Hydra website, http://www.hydra-robot.com/

[24] M. L. Shik, F. V. Severin, G. N. Orlovskii, *Control of walking and running by means of electrical stimulation of the mid-brain*, Biophysics, 11:756-765, 1966.

[25] S. Grillner, *Neurobiological bases of rhytmic motor acts in vertebrates*, Science, New Series, Vol. 228, No. 4696, pp 143-149, Apr. 12 1985.

[26] Ö. Ekeberg, *A combined neuronal and mechanical model of fish swimming*, Biological Cybernetics, 69, 363-374, 1993.

[27] The official Yamor's website, webpage: http://birg.epfl.ch/page53469.html

[28] D. Marbach, *Adam : A Modular Robot Evolution and Simulation Tool*, Unpublished Semestre Project, http://birg.epfl.ch/page32031.html, 2003.

[29] D. Marbach, *Evolution and Online Optimization of Central Pattern Generators for Modular Robot Locomotion*, Unpublished Semestre Project, http://birg.epfl.ch/page32031.html, 2004.

[30] Y. Bourquin, *Self-Organization of Locomotion in Modular Robots*, Unpublished Diploma Thesis, http://birg.epfl.ch/page53073.html, 2004.

[31] B. Mesot, *Self-Organization of Locomotion in Modular Robots: A Case Study*, Unpublished Diploma Thesis, http://birg.epfl.ch/page42735.html, 2004.

[32] P. Dürr, *Neubots, Simulation and Control of a Modular Underwater Robot*, Unpublished Diploma Thesis, 2004.

[33] T. Bertolote and V. Hentsch, *Design and prototyping of an underwater modular robot*, Unpublished Diploma Thesis, 2004.

[34] Russel Smith, *Open Dynamics Engine (ODE)*,
webpage: http://www.ode.org/.

[35] *GAlib*, webpage: http://lancet.mit.edu/ga/

[36] *The homepage of this project (containing amongst others the videos)*,
webpage: http://birg.epfl.ch/page56704.html

Appendix A

User/Programmer Manual

This chapter documents the main classes and methods which were developed for the framework. The first section describes the classes of the framework and their use and the second those from the Neubot simulator developed using the framework. For each the major methods are described but neither the simple getter and setter nor most private methods.

A.1 In-depth description of the framework

A.1.1 The classes

Refer to the Figure 5.1 for a valuable design diagram. All the protected or private variable finish by a "_" in the code. The general idea of the framework for the stepping is the following : a loop in the *main* method calls the step method on the *Simulation* which calls the step method on all the *Modules* that call the step method on his *Controller*.

Simulation.cpp

The creation and the management of the ODE world as well as all the objects into it is managed in the singleton *Simulation*. It contains the following major methods :

void **initialize**(dReal dt, Settings* settings)
Initialize all parameters and set up a new ode simulation world

void **step**()
Integrate the next time step in the simulation

void **draw**()
Draw all objects of the simulation

static void **nearCallback**(void *data, dGeomID o1, dGeomID o2)
The near callback function of dSpaceCollide()

Module.cpp

The abstract class *Module* contains all the generalistic informations about modules, among them positions, quaternion, number of blocks making up the module, or the reference to the controller.

Module(Picasso *pablo, Controller *c, int nbBodies, dVector3 *positions, dQuaternion *q, dVector3 *vel, dVector3 *angVel, int nb_sensors, Sensor *sensors)
Constructor of the module

virtual void **draw**() = 0
Draw the module

virtual void **step**(float dt) = 0
Update the state of the robot

virtual void **attach**(Module *m) = 0
Attach *this* module with m. The faces must be computed. This method must create a joint and call a method on m which alert it that a new joint was created

virtual void **deattach**(int f) = 0
Deattach *this* module from the module attached on the face f. Do exactly the contrary from attach(...)

GeomID **createTrimeshShape**(double* vertices, int vertexCount, int* indices, int indexCount)
Create the shape of the module by loading it from a ".obj" file generated for example by Wavefront.

Serializer.cpp

This abstract class offers a few methods to be implemented that must help to read and write structure of modules from and to a file (see A.2.2 for an example)

Serializer(string filename)
The constructor of the serializer. filename is the name from or to which we will write/read.

virtual void **getModules**(list<Module*> &listModules, Picasso *pablo) = 0
read the file and creates the modules and the joints between them and return them into listModules

virtual void **serialize**() = 0
writes the actual structure of the modules to the file specified before

Controller.cpp

This class must be sub-classed in order to implement a new controller.

Controller(int nbActuators)
The constructor which has one parameter specifying the number of actuators of the module.

virtual void **step**(float dt) = 0
Update the state of the controller

virtual void **getActuatorU**(double* u) = 0
return the power(s) (U) which must be applied on the actuator(s).

virtual void **setParameters**(double* params, int nbParams, bool* enable) = 0
for passing some optional parameters to the controller, or/and the flags array saying which actuators are enabled.

virtual void **setSensorValues**(double* params, int nbParams) = 0
gives to the controller the values of some optional sensor

Sensors.cpp

Represents all the sensors of one module. Contains only the values of this sensors in this generic implementation

GeometryLoader.cpp

This library is a free library customized by Peter Dürr. The shape of the modules can be created in a dedicated tool like Wavefront or Wings3D (free) and then exported in a .obj file which can be read by the library. The following code is an example of use of GeometryLoader.

```
// load the shape of the modules
CLoadObj* g_LoadObj = new CLoadObj(); // create a new loader
t3DModel g_3DModel;                   // create a new 3d model
g_3DModel.numOfMaterials = 0; // initialization of the 3d model
g_3DModel.numOfObjects = 0;
// launch the import of the shape
if(!g_LoadObj->ImportObj(&g_3DModel, "nameOfFile.obj"))
 cout<< "Error with the loading of the model of the module" << endl;
```

Then the number of vertex can be found with `g_3DModel.pObject[0].numOfVerts` and `g_3DModel.pObject[0].pVerts[i].x` return the coordinates x of the i-th vertex from the object 0.

Settings.cpp

This class is very useful to load dynamically some values from a file. The values are recorded into two maps. The first map contains *real* values instead of the second which contains *string* values (for the name of the files used in the application by example). The file must have the grammar described in Table A.1, where `anything` is any characters you want that would not be take into account, `real` is a real valued number with the usual syntax (scientific notation is supported) and `string` is any string of characters.

script	=	{line}
line	=	(variable | file) 'anything'
variable	=	'#' ident '=' 'real'
file	=	'$' ident '=' 'string'

Table A.1 : *Grammar for the settings file.*

The main methods are :

static Settings* **getInstance**()
Creates if it doesn't yet exist the singleton and returns it.

bool **initialize**()
Reads the file and fill the two maps.

void **setVariable**(string name, double value)
Manually changes the value of the variable named *name* or if it doesn't exist add it to the map containing the real variables.

double **getVariable**(string name)
Return the value of the variable named *name* if it exist else 0.0.

set/getFile(...)
These methods are the equivalent of the precedents methods but for the file's name map.

A.2 In-depth description of Neubot simulator

A.2.1 Design

The design is the same as on the figure 5.1 but instead of *NewSerializer* and *NewModule* it is *NeubotSerializer* and *NeubotModule*.

A.2.2 The classes

Simulation.cpp

We have adapted this class for the needs of our simulator. We added one callback method for the collision of the magnet's spheres :

static void **contactCallback**(void *data, dGeomID o1, dGeomID o2)
The callback function used to calculate the magnetic joint forces

NeubotModule.cpp

This is the implementation of the Neubot module. This is here that we calculate, into *step()*, the forces of the fluid dynamics. We calculated also on each step the positions of the 36 magnet's spheres.

The Controller

The differents controller developed are described in their respective chapter.

NeubotSerializer.cpp

We have elaborated a grammar for the serialization of Neubot structures (see Table A.2) where `generalities` contains the number of modules, their size, their mass and the number of joints of the structure. Then for each module, we give the position, the quaternion, the velocity, the angular velocity and an array of bit specifying wether the actuators are enabled or not. The modules are numbered from 1 to nbModules and in the order in which they appear. The joints are specified simply by saying which are the two modules to link.

Seth.cpp (main)

Seth is the name of all our simulator in reference to *Adam* from Daniel Marbach which is in some way the father of our own project. The main first load all the settings with the *Settings* class and put the default values for the variables. Then it create the simulation, initialize it and launch the serializer for getting the modules. It put these modules in the simulation. Finally, it initialize and launch an instance of Pablo and run it. That's all ! Enjoy !

script	=	generalities {module} joints
generalities	=	nbModules size mass nbJoints
nbModules	=	'#' 'integer'
size	=	'#' 'real'
mass	=	'#' 'real'
nbJoints	=	'#' 'integer'
module	=	position quaternion velocity angular_velocity enabled
position	=	'#' 'integer' 'integer' 'integer'
quaternion	=	'#' 'integer' 'integer' 'integer' 'integer'
velocity	=	'#' 'integer' 'integer' 'integer'
ang_velocity	=	'#' 'integer' 'integer' 'integer'
enabled	=	'#' 'integer' 'integer' 'integer' 'integer' 'integer' 'integer'
joints	=	{joint} *// the number of joint's must be the same as nbJoints*
joint	=	'#' 'integer' 'integer' *// the integers specifying the two modules that are connected*

Table A.2 : *Grammar for serialization of Neubot structures and coupling between them.*

A.3 Evolution of predefined structure

We used the same classes as previously for the evolution of the predefined structures, but we writed a new controller and a new serializer.

CPGSerializer.cpp

The grammar used to serialize or deserialize modules to or from a file is very similar as those we used previously in NeubotSerializer (see Table A.3). We just added the informations concerning the links between the oscillators. Each face of the modules which is actuated contains one oscillators. The oscillators are numbered from zero and in the order in which they appear.

CPGController.cpp

This controller is based on a PID controller and uses the output of the *CPG* object attached to each actuated face (cf Fig. B.2). The output of the neural network is multiplicated by a factor for amplification. This factor is usually 5 (cf Table A.4).

script	=	generalities {module} joints links
generalities	=	nbModules size mass nbJoints nbLinks
nbModules	=	'#' 'integer'
size	=	'#' 'real'
mass	=	'#' 'real'
nbJoints	=	'#' 'integer'
nbLinks	=	'#' 'integer'
module	=	position quaternion velocity angular_velocity enabled
position	=	'#' 'integer' 'integer' 'integer'
quaternion	=	'#' 'integer' 'integer' 'integer' 'integer'
velocity	=	'#' 'integer' 'integer' 'integer'
ang_velocity	=	'#' 'integer' 'integer' 'integer'
enabled	=	'#' 'integer' 'integer' 'integer' 'integer' 'integer' 'integer'
joints	=	{joint} *// the number of joint's must be the same as nbJoints*
joint	=	'#' 'integer' 'integer' *// the integers specifying the two modules that are connected*
links	=	{link} *// the number of link's must be the same as nbLinks*
link	=	'#' 'integer' 'integer' *// the integers specifying the two oscillators that are connected*

Table A.3 : *Grammar for serialization of Neubot structures containing oscillators.*

Variable	Value	Description
dt	0.0075	Timestep of the simulation
Physics Simulation		
g	9.81	Gravitation acceleration
rho	1000	Density of the Fluid
c_D	1	Dimensionless translation friction constant
c_T	1	Dimensionless rotation friction constant
Contact between modules		
p_magnet	9	Maximum force of the magnet in Newton
Neubot_MU	0.4	Friction coefficient of the neubot module
Neubot_EPSN	0	Bounce constant
Neubot_CFM	0.15	CFM of the modules
Neubot_ERP	0.2	ERP of the modules
jointForceLimit	11	resistance of the joints. Above this value the joint is broken.
World_CFM	0.001	CFM general
World_ERP	0.2	ERP general
Motors		
KP	0.1	P
KI	0.0001	I
KD	0.07	D
force	5	The response of the controller is multiplicated by this factor
Genetic algorithm		
encode_pmut	false	true if we want to evolve the mutation's parameters during coevolution
max_depth	variable	maximum depth of the tree genome
max_modules	variable	maximum number of modules of the robots
init_modules	variable	maximum number of initial modules of the robots
type_osc	2 or 3	2 : oscillators. 3 : continuous rotation.
rot_vel	0.4	magnitude of the torque applied on the module in direction of the hinge axis if type_osc == 3. As the torque is applied on the 2 modules in contrary sens, the resulting torque is 2*rot_vel, i.e. here with 0.4 it is 0.8 ∼PI/4 / second

Table A.4 : *Summary of the variables used in the simulator and their value during the second part of the project. These values have been chosen manually and needs maybe to be changed for another purpose. We have loaded them using the Settings class.*

Appendix B

Some Designs

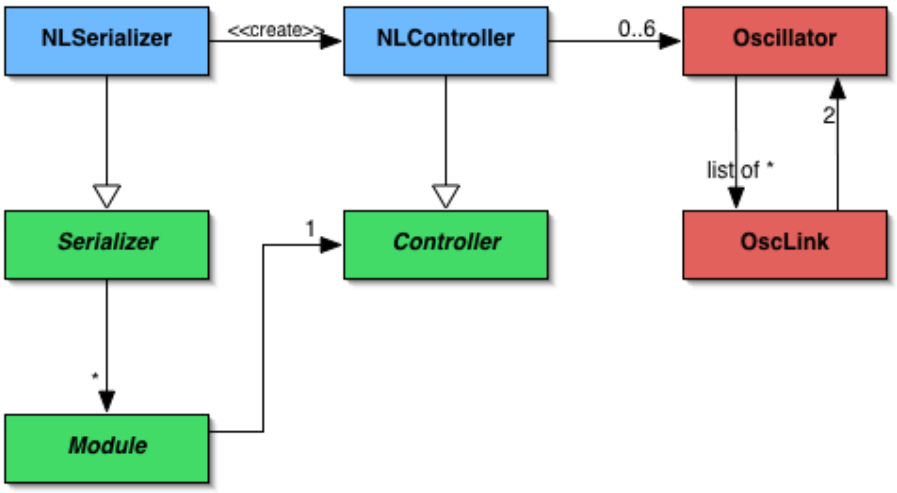

Figure B.1 : *The design of the non-linear oscillators with a part of the simulator.*

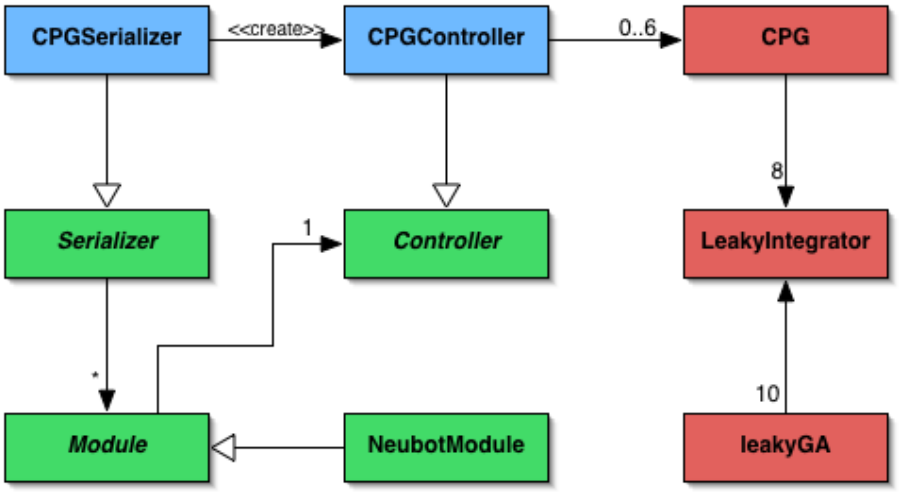

Figure B.2 : *The Design of the oscillator based on the leaky integrators with a part of the simulator. "leakyGA" is the main program containing the genetic algorithm to evolve the 8 neurons oscillator plus 2 tonic input designed as 2 neurons (10 neurons altogether). "CPG" is the class corresponding to the oscillator found by the GA. The neurons of type B have been removed.*

Appendix C

Description of the programs

A lot of programs have been developed during this project. In order to help the user (indeed the names are... very not helpful), they are all described in this appendix. For each, we answer the same three questions : What is it? What is the command line in order to compile? How to launch the program?
The sources are available at http://birg.epfl.ch/page56704.html and a XCode project file is given with them which avoid to use compile commands of five lines! We assume that you downloaded the sources and that you kept the directories as initially (five directories : picasso, Seth, GA, GA_Co, Leaky). Moreover, you need the following libraries: GAlib, GSL, ODE, Picasso, GLUT. I installed them in directory ∼/soft/lib/.

Seth

What is it? The initial program which allowed to create a simulation with or without graphical interface for simulate a modular robot described in a Structure file, with parameters loaded from a Settings file and whose module's shape is loaded from a *.obj* file. Why its name is *Seth* ? because we used the Adam simulator of Daniel Marbach as a base for our own simulator and the third son of Adam and Eve was named Seth...

What is the command line in order to compile? `cd Seth;`

```
g++ -Wall -o essai.exe picasso/*.cpp *.cpp ∼/soft/lib/libode.a -lGL -lGLU
-lglut -L/usr/X11R6/lib -lXi -lXmu
```

How to launch the program? `./essai.exe`
with valids Settings, .obj and structure files.

ga

What is it? This program is a genetic algorithm which uses the GAlib, and which launches the *Seth* program for each simulation and receive the resulting fitness value thanks to a file.

What is the command line in order to compile? `cd Seth;`

```
g++ -Wall -o GA ga.cpp -lode -I/home/vonhalle/soft/include
/home/vonhalle/soft/lib/libga.a Settings.cpp io.cpp
```

How to launch the program? `./GA`

NLSeth

What is it? Same program as *Seth* but with a different controller which use a non-linear oscillator coupled with those of the other modules.

What is the command line in order to compile? `cd Seth;`

```
g++ -Wall -o NLSeth ../GA/NLSeth.cpp ../GA/OscLink.cpp ../GA/Oscillator.cpp
../GA/NLController.cpp ../GA/NLSerializer.cpp *.cpp picasso/*.cpp
~/soft/lib/libode.a ~/soft/lib/libgsl.a -I/home/vonhalle/soft/include
-I/home/vonhalle/Dev/Seth -lGL -lGLU -lglut -L/usr/X11R6/lib -lXi -lXmu
```

How to launch the program? `./NLSeth`

cpg_ga

What is it? Same program as *ga* but which launches simulation with modules that use non-linear oscillator as controller.

What is the command line in order to compile? `cd Seth;`

```
g++ -Wall -o cpg_ga ../GA/*.cpp *.cpp picasso/*.cpp
-I/home/vonhalle/soft/include -I/home /vonhalle/Dev/Seth
/home/vonhalle/soft/lib/libga.a /home/vonhalle/soft/lib/libgsl.a
~/soft/lib/libode.a -L/usr/X11R6/lib -lXi -lXmu -lGL -lGLU -lglut
```

How to launch the program? `./cpg_ga`

LeakyTest

What is it? Little program which create a neural network with eight leaky integrators and couples them with the informations of leakyGenom.txt.

What is the command line in order to compile? `cd Leaky;`

```
g++ -Wall -o leakyTest leakyTest.cpp LeakyIntegrator.cpp
```

How to launch the program? `./leakyTest`

LeakyGA

What is it? This program evolve the connections between eight neurons. I use it to get the neural oscillator described in chapter 9.2.

What is the command line in order to compile? `cd Leaky;`

```
g++ -Wall -o leakyGA leakyGA.cpp -I/home/vonhalle/soft/include
/home/vonhalle/soft/lib/lib ga.a LeakyIntegrator.cpp
```

How to launch the program? `./leakyGA`

cpgTest4

What is it? This program is the simulator, but this time with a controller based on the neural oscillator (CPG) and with a new (de)serializer.

What is the command line in order to compile? `cd Leaky;`

```
g++ -Wall -o cpgTest4 cpgTest4.cpp CPG.cpp LeakyIntegrator.cpp
CPGController.cpp CPGSerializer.cpp ../Seth/ picasso/*.cpp ../Seth/*.cpp
-I/home/vonhalle/soft/include -I/home/vonhalle/Dev/Seth
-I/home/vonhalle/Dev/Leaky ~/soft/lib/libode.a -lGL -lGLU -lglut
-L/usr/X11R6/lib -lXi -lXmu -O3
```

How to launch the program? `cpgTest4 -structure file.txt -nparams X` with `file.txt` the file containing the structure which we will deserialize and `X` the number of genes in the genome (depending on the structure).

cpgTestga1

What is it? This program is a genetic algorithm which uses *cpgTest4* as simulator for the robots whose structure are recorded in a file. I used this program to evolve the locomotion ability of predefined structure (cf chapter 10).

What is the command line in order to compile? `cd Leaky;`

```
g++ -Wall -o cpgTestga1 cpgTestga1.cpp ../Seth/io.cpp ../Seth/Settings.cpp
-I/home/vonhalle/soft/include -I/home/vonhalle/Dev/Seth
-I/home/vonhalle/Dev/Leaky /home/vonhalle/soft/lib/libga.a ~/soft/lib/libode.a
-L/usr/X11R6/lib -lXi -lXmu -lGL -lGLU -lglut -O3
```

How to launch the program? `./cpgTestga1`
and all the parameters needed. See the source for knowing which are the line command parameters.

CPG_Seth

What is it? This program implements the co-evolution algorithm (cf Chapter 11).

What is the command line in order to compile? `cd Leaky;`

```
g++ -Wall -o cpg_Seth_GA CPG_Seth.cpp CPG6.cpp LeakyIntegrator.cpp
RotController.cpp CPGController.cpp CPGSerializer.cpp ../Seth/picasso/*.cpp
../Seth/*.cpp ../GA_Co/*.cpp -I/home/vonhalle/soft/include
-I/home/vonhalle/Dev/Seth -I/home/vonhalle/Dev/Leaky
-I/home/vonhalle/Dev/GA_Co -I/home/vonhalle/Dev/Seth/picasso
/home/vonhalle/soft/lib/libga.a ~/soft/lib/libode.a -lGL -lGLU -lglut
-L/usr/X11R6/lib -lXi -lXmu -O3
```

How to launch the program? `./cpg_Seth_GA`
and all the parameters needed. See the source for knowing which are the line command parameters.

launcher

What is it? A little program which launches *CPG_Seth* and restart it if it crashes (never seen) or if it consumes too much memory.

What is the command line in order to compile?

```
g++ -Wall -o launcher
-I/home/vonhalle/Dev/GA_Co launcher.cpp
```

How to launch the program? `./launcher`
and all the parameters needed. See the source for knowing which are the line command parameters.