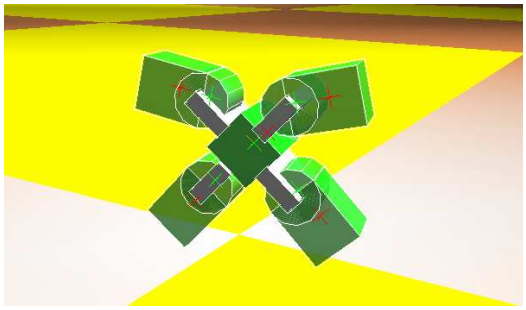
MSc Dissertation

Self-Organization of Locomotion in Modular Robots

Yvan Bourquin
Candidate No 81214 for MSc in Evolutionary and Adaptive Systems
mail@yvanbourquin.com
Department of Informatics
University of Sussex, Brighton, United Kingdom



EPFL/BIRG Supervisor
Prof. Auke Jan Ijspeert
Swiss Federal Institute of Technology

BIOLOGICALLY INSPIRED
ROBOTICS GROUP (BIRG)

Sussex Supervisor
Dr. Inman Harvey
University of Sussex

US University of Sussex

Abstract

The Biologically Inspired Robotics Group (BIRG) of the Swiss Federal Institute of Technology in Lausanne (EPFL) is currently developing new hardware and software for an experimental modular robotic platform. Modular robots are robots built of several similar building blocks called *modules*. Usually modules have one or two hinges that allow movement with one or two degrees of freedom. Locomotion of the robot, e.g. crawling, is possible when several modules are put together. By means of computer simulations, this study explores the self-organization of locomotion in various assemblage configuration of the BIRG's robot. Locomotion is enabled by the regular oscillations of the robot's servomotors, which are produced by Central Pattern Generators (CPG) inspired from the nervous system of vertebrates. Optimal oscillatory parameters are found using different numerical search methods: *genetic algorithms*, *simulated annealing*, *particle swarm optimisation* and *random search*. Although all these methods generate efficient locomotion patterns, the best results are obtained by *particle swarm optimization*.

Acknowledgements

I would like to thank the staff from the Biologically Inspired Robotics Group of the EPFL for their help with this project. Especially I would like to thank my supervisor Auke Ijspeert for his ideas, for the time he spent in my supervision and in the correction of my draft. Thanks also to Olivier Michel for his constant improvements in the robot simulator Webots. Thanks to Joel for the discussions on PSO and other stuff. Thanks to Alessandro for being the most attending system administrator of all time. Thanks to Jonas for his explanations on oscillators, his Matlab hints and the discussions about other bio-inspired subjects. Thanks very much to Tasneem for proofreading this manuscript. Thanks to Rico and Elmar for their work on the robot's hardware and for the good mood in the lab. Thanks also to Inman Harvey for the "remote" supervision, comments and review of the draft. Finally, thanks to Larisa for tolerating my boring computer geek's discussions, my late work schedules and my computer humming 24 hours a day during simulations.

Table of content

1 Introduction

From the outset, artificial intelligence was mainly concerned with logic and mathematical problems. When computers were first developed, researchers were theorists or mathematicians who were naturally interested into problems based on pure logic such as game playing programs. Within a short time, computers became able to outperform humans in logical operations, through their ability to achieve fast computations. Therefore, the scientists predicted that artificial intelligence would surpass human intelligence within decades.

In that euphoric time, problems such as locomotion and vision were considered auxiliary or trivial given that any child or animal can move about and see without even thinking about it. Scientists had successfully created intelligent programs that could outperform animals or humans in some particular tasks. The tasks that these computer programs were able to solve were of higher intellectual concern for the scientists and therefore they thought that the artificial intelligences they had just created were already superior to natural intelligence.

However, very soon, people started to think that artificial intelligence should also be used for purposes that were more practical. Controlling robots for industrial or domestic tasks, as seen in science fictions, was seen to be a real possibility. The first essential step for a useful robot is the ability to move about and therefore, research started to investigate locomotion. However, the first prototypes based on logical principles were not very successful. For example, like the famous Shakey, early mobile robots were programmed to build internal 2-d or 3-d geometrical representations of their surrounding environment. However, the poor processing performance of the processors of that time, were challenged by the complexity of the task. The robots spent most of the time computing and moved very slowly. The scientists realised that any animal could do better than their machines. They started to understand that the problems they considered the easiest were actually the most difficult.

With Rodney Brooks [Brooks, et al. 1989] came a groundbreaking period during which the rightfulness of the traditional *sense-model-plan-act* architecture was questioned. Legged robots based on his famous *subsumption* architecture appeared. They showed strikingly fast insect-like locomotion and behaviour. The subsumption architecture marked a turning point from traditional artificial intelligence to more bio-inspired concepts.

In the nineties, teams from the Swiss Federal Institute of Technology and University of Sussex [Harvey, et al. 1997] formulated the principles of a new field: *evolutionary robotics*. Artificial neurons inspired from their natural counterpart were used together with evolutionary techniques to develop artificial nervous systems. With evolutionary robotics, it became possible, without human design, to obtain efficient wheeled or legged locomotion, obstacle avoidance and other forms of behaviour in real or simulated robots.

Nevertheless, real robots are made of mechanical and electronic parts. Therefore, only the evolution of the control circuitry, implemented in software, could be experimented upon. However, in addition to the nervous systems, in nature, the animal morphologies are subject to evolution as well. It was therefore, interesting to see how morphologies would develop under artificial evolution. Therefore, Karl Sims [Sims 1994] developed the first computer simulation of the evolution of bodied creatures in a virtual physics environment.

1.1 Objectives

The objective of this project is to explore locomotion in an experimental robotic platform developed at the Biologically Inspired Robotics Group (BIRG). Locomotion in modular robots is generally achieved through simple state automata. In this study, a radically different approach is investigated: the robot must discover efficient locomotion patterns autonomously in computer simulations. Non-linear oscillators are used as motor control signals. The oscillators' parameters are optimized through numerical methods including *genetic algorithms*, *simulated annealing* and *particle swarm optimization*. In addition to obtaining efficient locomotion gaits, the objective is to compare the performance of these various search methods. For this reason, several predetermined configurations of the BIRG's robot were designed and simulated in an ODE-based physics simulator.

1.2 Modular robots

Modular robots are robots build of multiple identical building blocks called *modules*. The idea behind modular robotics is freely inspired by cellular automata and by social insects. Through cooperation, social insects, such as ants, bees or termites, achieve feats that would be impossible to achieve by a single individual. For example, ants can use their own body as bridge to allow other ants to move safely above a crag (Figure 1). By using similar principles, modular robots perform *self-reconfiguration,* e.g. they autonomously change their shape and adapt to different kinds of terrain. For example, some modular robots can transform into a snake to tunnel through a pipe then later transform into a quadruped to go up stairs or even climb a fence. Often, the shapes are inspired by animals but loop and wheel configurations are also possible.

Figure 1: Ants forming a bridge.

Another advertised property of modular robots is *self-reparation*. Because the modules of modular robot are identical, it is possible to excise a damaged module and replace it with a spare one, if it is available. Like their biological counterparts, the modular robots accomplishments are made possible because they are built upon redundant and decentralized architectures. However, modular robots frequently are not fully decentralized and do not really

qualify as *self-organized* systems. This is because, in most cases, one module is used as the master and the others as slaves, although these roles are interchangeable.

Although this area has progressed rapidly since its beginning in the early nineties, modular robots have only reached their goal in very controlled laboratory environments. Potential future applications of modular robots include locomotion and self-assembly in human-unfriendly environments.

1.3 Existing projects

1.3.1 M-TRAN

One of the most accomplished modular robotic projects is the M-TRAN (Figure 2) developed by the Distributed Systems Design Research Group of the AIST in Japan [Kamimura, et al. 2001]. M-TRAN is a modular robot able to self-reconfigure without human intervention. M-TRAN modules are docked through permanent magnets and disconnected from each other by heating a Shape Memory Alloy (SMA) coil that releases a force in the opposite direction as the permanent magnets. Each M-TRAN module has two motors which provide it with two degrees of freedom in the same 3d-plane.

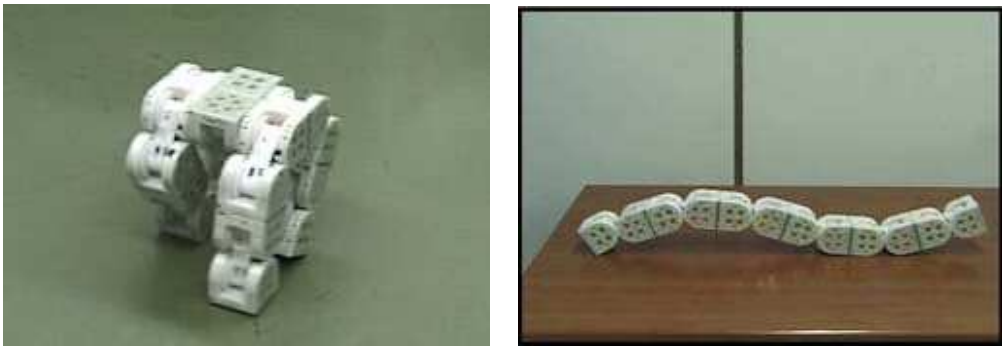


Figure 2: M-TRAN II configured as quadruped and caterpillar.

1.3.2 PolyBot

The XEROX Palo Alto Research Centre is running several research projects in modular robotics. Their PolyBot project (Figure 3, left) uses modules able to self-reconfigure and dock also using shape memory alloy. The modules are hermaphroditic and can be docked to each other at four different 90° rotation angles. However, each module has only two connection surfaces (Figure 3, right) and therefore mostly snake-like configurations are possible unless passive elements are used. PolyBot Generation 3 modules also have joint angle sensors, accelerometers and infrared proximity sensors used principally to aid in docking two modules. Control is centralized using gait tables.

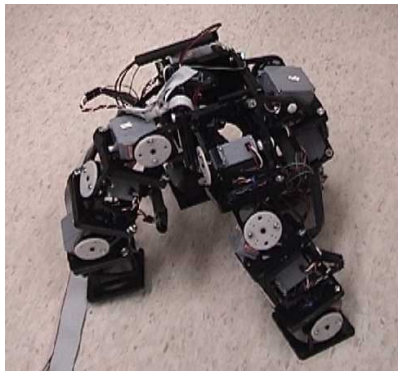



Figure 3: Left: PolyBot Generation 1 with 16 modules in a four-legged spider configuration. Right: PolyBot Generation 3 module.

1.3.3 CONRO

The CONRO project is taking place at the Information Science Institute of the University of Southern California. Each CONRO modules has two degrees of freedom, in two different axes, which is particularly helpful for legged locomotion, but makes docking a challenge as compared to the M-TRAN and PolyBot. The docking of two modules is achieved through infrared communication; it requires several seconds. 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 though a hormone inspired decentralized communication system that supports servo-commands and on-line reconfiguration [Shen, et al. 2002]. Much like natural hormones, artificial hormones allow different types of responses from different parts of the robot's body.

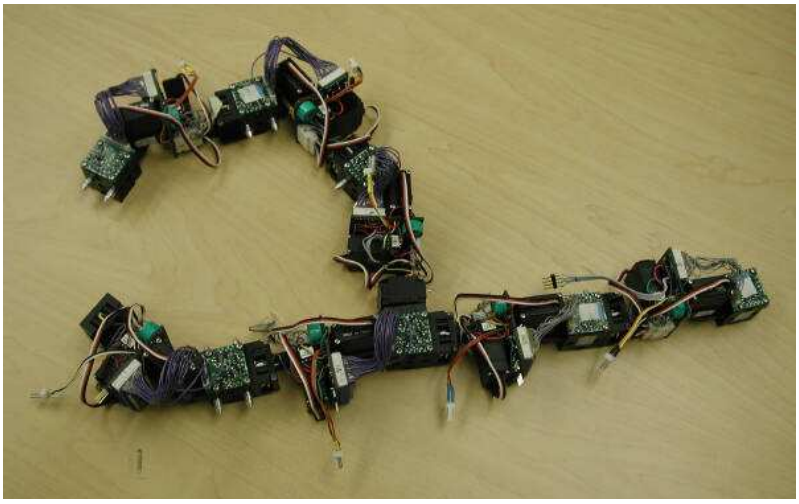


Figure 4: CONRO (USC Information Sciences Institute).

Most of the research performed on modular robotics has focussed on self-reconfiguration or locomotion. Many of these robots do not have a large palette of sensory capability and therefore their use in investigating artificial intelligence is limited.

1.3.4 Other projects

Lattice robots are also modular robots but are based on a very different architecture. They are built of square or cubic modules that use linear actuator to reconfigure or move, instead of rotational ones. Telecube [Suh, et al. 2002] is an example of cubic lattice robot. Telecube uses "telescoping-tube linear actuator", which consist of motors and lead screw in a housing, to extend or collapse its six faces. Docking is accomplished using "switching permanent magnets latch".

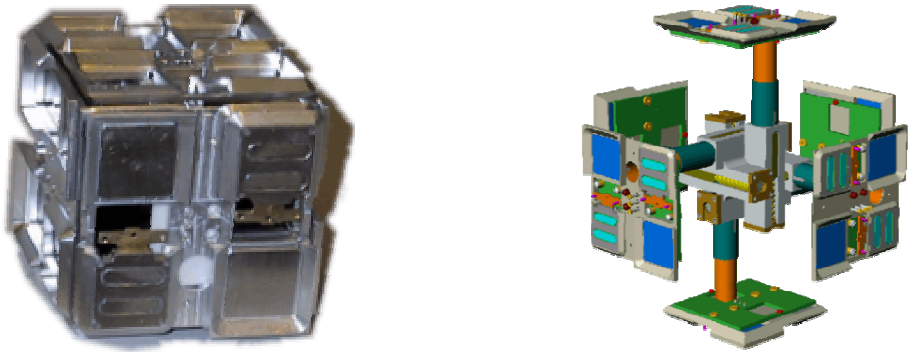


Figure 5: Telecube module (Palo Alto Research Centre)

Locomotion of lattice robots works either by sequences of contraction and extension or by a reconfiguration that shift the gravity centre. In both cases, it is quite slow compared to other types of modular robots.

1.4 BIRG's robot

The BIRG's modular robot is a recent project; it is not as technically advanced as the projects described above; in particular, no definitive docking mechanism has yet been designed. However, it does have some interesting features: First, inter-module communication is achieved through wireless communication via Bluetooth. This allows communication between modules that are neither connected nor visible to each other.

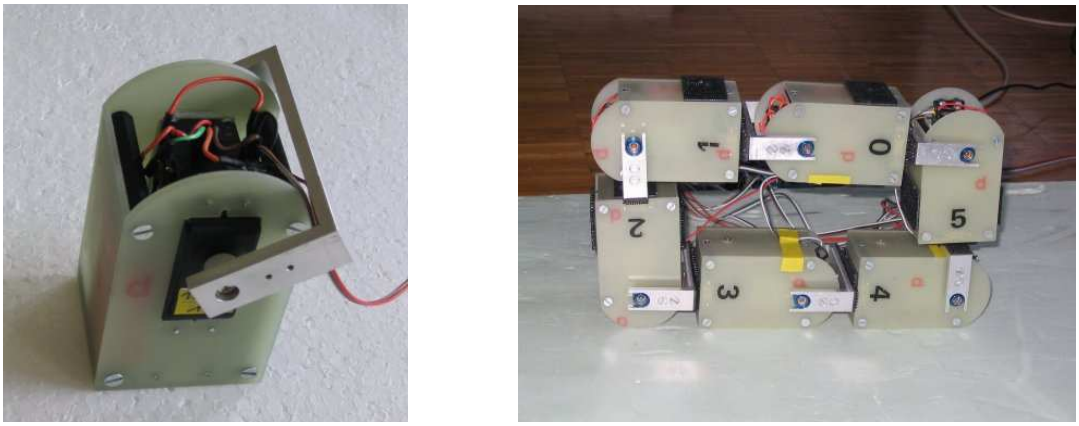


Figure 6: Left: BIRG module. Right: the BIRG robot in a loop configuration.

Thanks to the wireless communication, a new paradigm appears. It is potentially possible to disjoint the robot into several subunits which can still operate in accord and which can later

assemble and operate as whole again. In addition, the BIRG's modules are controlled by Field Programmable Gate Arrays (FPGA). These allow faster and more generic computation than the usual micro-controllers. The dimensions of the BIRG's modules are 87 x 50 x 45 mm (*height* x *width* x *depth*, see Figure 6). A module weights around 250 grams, and its motor force is 73 Newton · cm, which is more than usually encountered in modular robots and roughly allows one module to lift two others. The hinge can turn in a range somewhat larger than 180º.

At the time of writing, the BIRG robot does not feature any sensing modality and therefore, the present discussion is restricted to the study of locomotion without sensory feedback.

1.5 Locomotion by oscillations

Locomotion is an essential skill in animals. It is required for hunting preys, escaping predators, or more generally, finding food and mates. Locomotion is achieved by applying forces on a terrestrial, aerial or aquatic environment. These forces are generated by the rhythmic contraction of muscles attached to limbs; wings, legs, fins and so on.

This document is concerned only with locomotion on a flat ground surface and under gravitational force. A locomotory gait is efficient when all the involved muscles contract and extend with the same frequency[1]. Typical terrestrial legged locomotion gaits are *walk*, *trot* and *bound.* According to the gait, different activation phases for the different limbs are required. For example, a quadruped's *walk* consists normally of four different phases, e.g. the feet hit the ground at four different times. In the example (Figure 7a.): first the left hind leg (LH) hits the ground, then the left foreleg (LF), then the right hind leg (RH) and finally, the right foreleg (RF), then the sequence repeats. All legs share a common frequency but different phases.

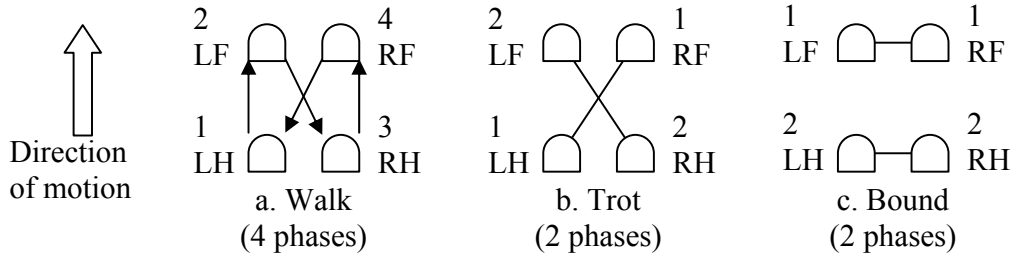


Figure 7: Three different quadruped gaits.

Trot and *bound* have only two different phases (Figure 7b. and 1c.). Trotting is when the two diagonally opposed legs reach the ground simultaneously. Bounding is when in a first time the forelegs hit the ground together, then in the second time, the hind legs hit the ground, also together. In biped, quadruped, or hexapod robots, locomotory gaits are generated using these simple principles: same frequency, different phases.

[1] Note that this is not always true; in some cases gaits can also be composed of muscle movements where frequencies are not equal but are multiples of each another.

1.6 Controllers architectures

Numerous different architectures have been tried for robotic locomotion. Let us review shortly the principal ones.

1.6.1 Genetic programming

In his pioneering work (Figure 1), Karl Sims' [Sims 1994] virtual creatures were controlled using a kind of genetic programming, where each neuron could carry out a different mathematical function. Some of the available function could produce oscillations from constant inputs and therefore this was very suitable for locomotion even though it was not a very biologically plausible approach.

Figure 8: Virtual Creatures competing for the green block [Sims 1994].

1.6.2 Neural networks

With *evolutionary robotics*, a new concept appeared, originating from both the University of Sussex [Harvey, et al. 1997] and the Swiss Federal Institute of Technology [Mondada, et al. 1995]. In this approach, robots are controlled with neural networks whose parameters are optimised using genetic algorithms. Many different types of neural networks have been experimented upon, for example feed-forward or recurrent architectures using sigmoid transfer function.

Randall Beer [Beer 1996] introduced a new type of neuron using a time constant: Continuous Time Recurrent Neural Network (CTRNN). Beer and his team used CTRNNs for simulating cognitive behaviour and among other things for the control of an autonomous hexapod robot [Gallagher, et al. 1996]. In this article, CTRNN controllers were evolved in simulation using genetic algorithm and the results were transferred directly to a hardware robot. The controller's neurons were implemented directly in electronic components such as amplifiers and resistors.

At Sussex, Nick Jacobi [Jacobi 1998] developed, in simulation, a control system for an octopod robot (Figure 9) using the CTRNN techniques. His "minimal simulation" methodology allowed the successful transition of the controller into a real robot able to wander around and avoid obstacles using infrared sensors, bumpers, whiskers and light sensors.

Figure 9: The Octopod robot.

Plastic Neural Network (PNN) [Floreano, et al. 1996] changed the classical leaning concept of evolutionary robotics. With the PNN approach, a part of the learning process is transferred to the robot's lifetime. More precisely, the genetic algorithm selects the neurons' learning rules but not the synaptic connections weights, which are modified during the robot lifetime according to various genetically encoded leaning rules.

With GasNets [Husbands, et al. 1998] the concept of synaptic weights itself is turned upside down. A new mode of transmission based on simulated neurotransmitters is superimposed to the neural network. These neurotransmitters are able to change the intrinsic properties of the neurons and therefore a new kind of plasticity is enabled.

1.6.3 Oscillators

Although CTRNNs can be used to obtain the oscillations required for locomotion, an approach that consists in using non-linear oscillators seems simpler. With non-linear oscillators, the oscillatory behaviour can be assumed and it is possible to focus on the problem of the oscillators' interconnection.

Furthermore, a lot of work has been carried out using oscillators. From Taga's [Taga 1994] research on bipedal locomotion to Kimura quadruped robots walking on irregular terrains [Kimura, et al. 1998] and with Ijspeert's work on aquatic and terrestrial locomotion in salamander [Ijspeert 2001], non-linear oscillators have been applied many times successfully to locomotion problems.

However, we are aware of only two projects in which non-linear oscillator are applied to the locomotion of modular robots [Mesot 2004, Yoshida, et al. 2003]. In the latter article, the authors present two different methods based on genetic algorithms that explore locomotion in the M-TRAN robot. Their first method obtained locomotion by applying genetic algorithms to the robot reconfiguration sequence. Their second method is similar to this project; genetic algorithms are applied to the optimization of the parameters of non-linear oscillators to evolved locomotory gaits.

2 Methods

2.1 Simulator

The simulations of this study were carried out on a commercial robot simulator called Webots™ [Michel 2004], which is developed by Cyberbotics Ltd in collaboration with the EPFL. Webots™ is built on ODE (Open Dynamics Engine) and it allows therefore realistic physics simulation. With Webots™, the robots structure and its environment are built using a VRML editor. The robot controllers must be programmed in a high-level programming language such as C, C++ or Java.

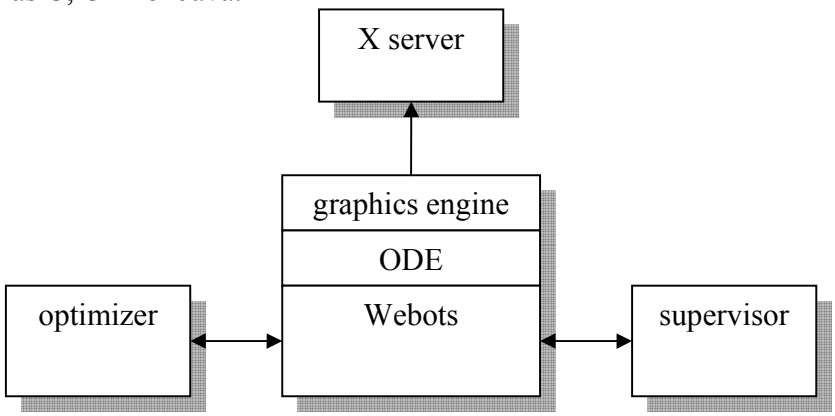


Figure 10: Software architecture.

For the simulations described here, two controllers were developed. The first controller is the "optimizer", which does most of the work: it runs the search algorithms, computes the oscillations, sends servomotor commands to the simulator and measures the performance. The second controller is the "supervisor*"* whose role is just to move robots to their start position at the beginning of each evaluation.

2.2 Experimental morphologies

A crucial point was the choice of good morphologies for the simulations. To ensure the robustness of the results, we decided to employ four different configurations. The initial thought was to investigate randomly assembled structures; however, after the first experiment, it became clear that bio-inspired morphologies would be more interesting because the resulting locomotion gaits can be compared with their natural counterparts.

The first morphology, the *wheel* (Figure 11, left), is a simple four-ways symmetrical robot made of five modules. The middle module is placed such that its mass centre is located in the same 3d-plane as the other ones; this decreases the robot's chances of tipping over. The second morphology, the *caterpillar* (Figure 11, right) is made up of six modules whose axes of rotation are all in a vertical plane.

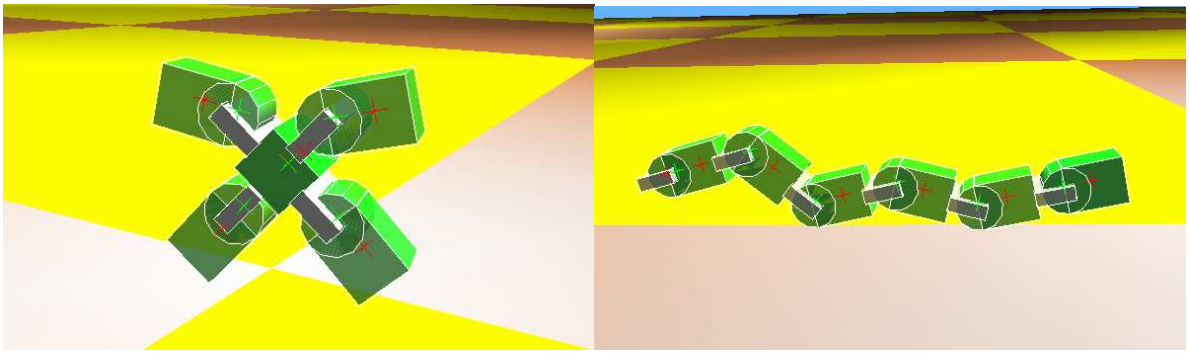


Figure 11: Wheel and caterpillar configurations.

The *tetrapod* (Figure 12, left) is a four-ways symmetrical structure made of nine modules; it has four legs and a central passive module. Finally, the *crawler* (Figure 12, right) is built out of 12 modules. It is left/right symmetrical like real quadrupeds. The crawler's "hip" joints move horizontally and the "knees" joints move vertically. Vertical "hips" and horizontal "knees" would have been another design option. The crawler's body motors are deactivated and therefore its spine remains straight.

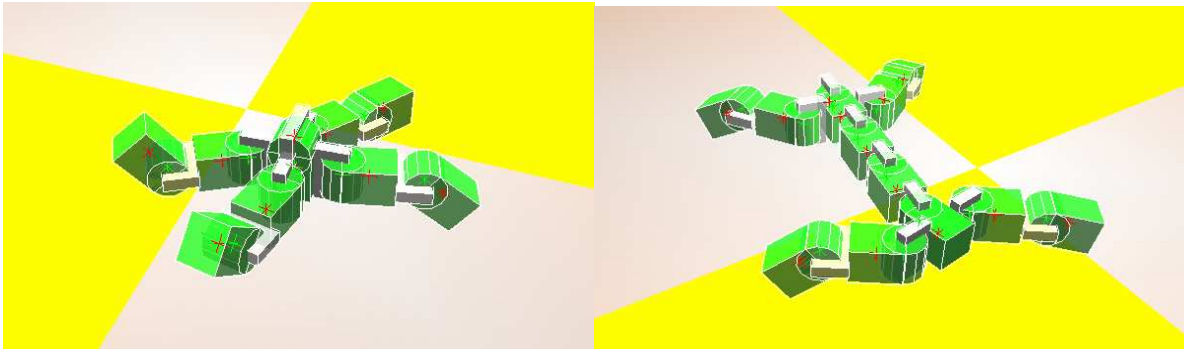


Figure 12: Tetrapod and quadruped configurations.

2.3 Oscillators

The present chapter (2.3) summarizes the principles used for locomotion using non-linear oscillators, based on previous work at the BIRG.

In animal locomotion, the oscillations of the joint angles produced by the muscular activity can have different waveforms. These waveforms are usually smooth: brutal transitions are uncommon. In order to facilitate numerical simulations, a strong simplification is to model locomotion as sinusoidal variations of the robot's joint angles. As seen before, these sine waves must have the same frequency but they can differ in phases and amplitude, according to the gait. In other words, a module's motor activation can be controlled by this simple equation:

$$x(t) = A \sin(\omega t + \varphi) + x_0 \qquad (1)$$

where $x(t)$ is the desired servomotor angle of a module at time t, A is the amplitude of the oscillations, φ is the oscillation phase, and x_o is the angular midpoint of the oscillations of the module.

In practice, sinusoidal signals are not flexible enough, because they do not allow a soft transition from one gait to another. For example, if a walk gait in a robot is controlled by sinusoidal motor signals, the transition to a different gait, say trot, requires the activation of different oscillations' phases, amplitudes and frequencies. However, with sinusoidal signals, the transition from one gait to another is brutal and therefore cannot be carried out satisfactorily by the robot's motors. Consequently an uncontrolled transition appears, during which the robot performs an undesired brutal movement and is subject to fall. This is in contrast to gait transitions in nature, which always occur smoothly. Furthermore, with sinusoidal signals, there is no simple way to incorporate sensory feedback.

2.3.1 CPGs

The concept of using non-linear oscillators to control robotic locomotion is inspired from biology. Experiments [Shik, et al. 1966] showed that, in a decerebrated cat, the electrical stimulation of the brainstem is able to induce walking. Furthermore, an increase of the signal strength changes the walk velocity and the transition from a walking to a trotting gait happens autonomously. These experiments demonstrated that the brain is not involved in the generation of the rhythmic signals that produce locomotion in the cat.

Grillner [Grillner 1985] explained that the locomotory signals that produce sequences of muscle activation, such as walk, trot or gallop are generated by Central Pattern Generators (CPG) located in the spinal cord. These CPGs are neural circuits that generate oscillatory output from a tonic input coming from the brain. The brain appears to play a higher-level role such as regulating the initiation, velocity and termination of the locomotory activity.

2.3.2 Non-linear oscillators

As we have seen, the sine approach of equation (1) does not enable smooth gait transitions. To overcome this problem, non-linear oscillators were introduced as mathematical models of the natural CPGs [Ijspeert, et al. 2003]. The state of oscillators changes smoothly and therefore, gait transitions are soft. In addition, with oscillators, feedback can be incorporated in the simulation. For example, sensors can detect that a foot is in contact with the ground and a feedback signal can be injected into the oscillators.

The oscillator proposed in [Ijspeert, et al. 2003] is based on these differential equations:

$$\tau \dot{v} = -\alpha \frac{x^2 + v^2 - E}{E} v - x \qquad (2)$$

$$\tau \dot{x} = v \qquad (3)$$

where v and x represent the current state of the oscillator, E is a positive constant that represents the energy of the oscillator, α determines the rate of convergence towards the limit cycle and τ is the time constant that determines the oscillation's frequency. This type of oscillator converges to a sinusoidal signal with amplitude $\sqrt{E}$ and period $2\pi\tau$ [Ijspeert, et al. 2003]:

$$\tilde{x}(t) = \sqrt{E} \sin(t/\tau + \phi) \qquad (4)$$

where ϕ depends on the initial conditions. This behaviour is illustrated by the limit cycle in Figure 13, which represent the results of 30 oscillations started with random initial

conditions x and v in the range [-2, 2]. The parameters used were $\alpha = 0.7$ and $E = 1$. As it can be observed each run converges to the circular attractor of diameter $2\sqrt{E}=2$ (see Matlab code in the appendix).

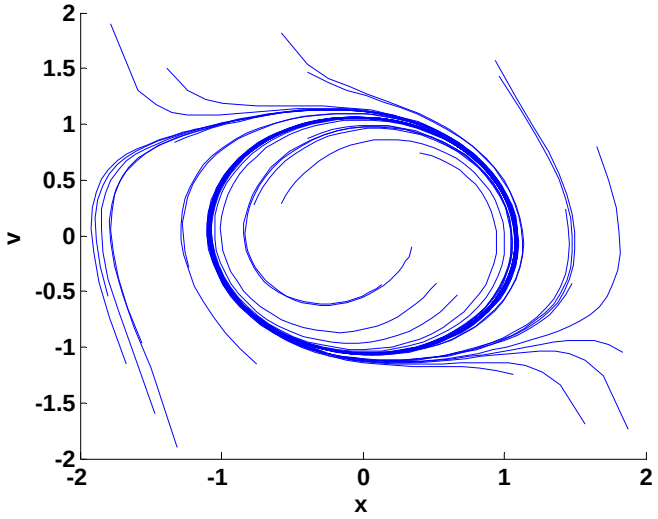


Figure 13: Limit cycle of a standalone oscillator.

2.3.3 Oscillator synchronization

By choosing the E and τ parameters, it is possible to control the amplitude and frequency of the oscillations. However, locomotion is efficient only when the phase shifts between the oscillations stays constant through time and therefore, a strict synchronization is required. In the model proposed in [Ijspeert, et al. 2003], synchronization is obtained by coupling the oscillators; a signal proportional to the sum of the state of every other oscillator is added into each oscillator. Equation (2) seen earlier, is now completed into equation (5) [Ijspeert, et al. 2003]:

$$\tau \dot{v}_i = -\alpha \frac{x_i^{\,2} + v_i^{\,2} - E}{E} v_i - x_i + \sum_j^N (a_{ij} x_j + b_{ij} v_j) \qquad (5)$$

$$\tau \dot{x}_i = v_i \qquad (6)$$

where a_{ij} and b_{ij} represents the strength of the coupling of the x and v states of oscillator j into the oscillator i.

Synchronization happens only when the uncoupled frequencies match approximately [Pikovsky, et al. 2001]. Figure 14 illustrates this fact: the frequency difference Δf of two uncoupled oscillators is plotted versus frequency detuning ΔF after coupling. If the uncoupled frequencies are too different, synchronization does not occur.

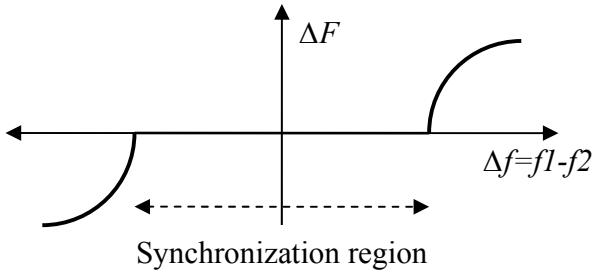


Figure 14: Frequency vs. detuning graph [Pikovsky, et al. 2001].

In order to facilitate synchronisation the same time constant τ is used for all the oscillators and therefore the uncoupled frequencies are similar. A frequency of 1Hz ($\tau = 1/(2\pi)$) is chosen as baseline for all simulations. This is because it corresponds to the pace of an ordinary animal and it is slow enough for the physical robot's servomotors to go once 180° back and forth. A stabilization period is necessary before the frequencies become locked. The duration of the stabilization period depends on the coupling strength.

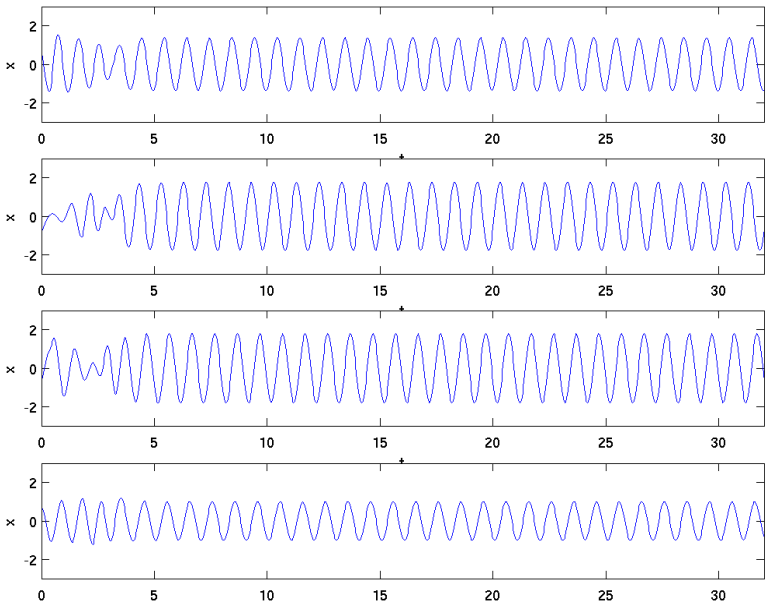


Figure 15: Example of synchronized oscillations.

Figure 15 shows an example of synchronized oscillations (see Matlab code in appendix). The x states of four coupled oscillators are plotted over a period of 32 seconds: the initial stabilization period is visible. This stabilization period is bad because it results in disorganized steps of the robot. Shorter stabilization periods are wished and can be obtained by increasing the coupling strength. However, unlike standalone oscillators, the signals produced by coupled oscillators are not exact sine waves. Discrepancy with the sine increases with the coupling strength and furthermore, when the coupling becomes too strong the signals turn out to be chaotic (Figure 16) and unsuitable for controlling locomotion. For that reason, coupling strengths are suitable only within an appropriate range that must be determined.

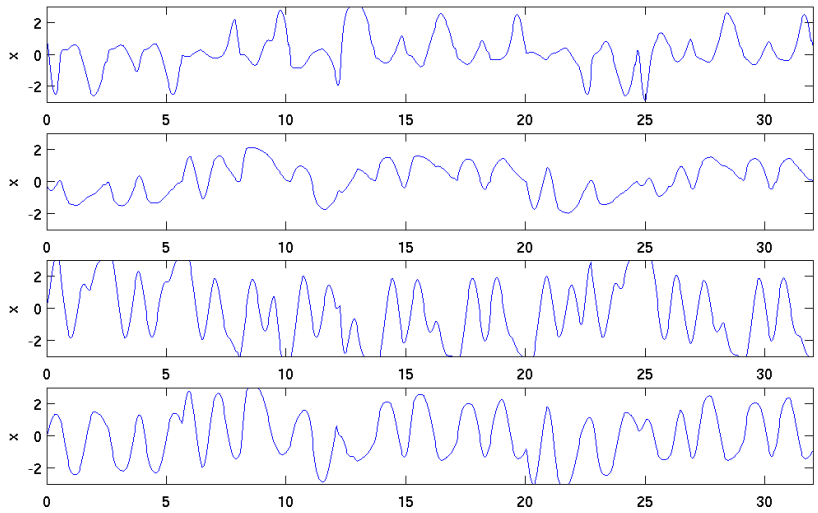


Figure 16: Chaotic oscillations resulting of too strong coupling.

When multiple coupled oscillators are used, the resulting phase shifts between the oscillations is a function of the coupling strengths a_{ij} and b_{ij}, however several different combination of a_{ij} and b_{ij} can produce the same phase shift. In fact, the exact outcome of a particular coupling combination cannot be predicted by any general theory [Pikovsky, et al. 2001]. Consequently, the coupling strengths must be optimized by the search algorithms.

A slight variation (7) of the original oscillators' model seen above was proposed in [Mojon 2004]. With this modification, the inputs are normalized and therefore, the "strength of the signal carried by a particular connection does not depend on the energy of the emitting oscillators" [Mojon 2004]. This modified model (7) is used in this project.

$$\tau \dot{v}_i = -\alpha \frac{x_i^2 + v_i^2 - E}{E} v_i - x_i + \sum_j^N \frac{a_{ij} x_j + b_{ij} v_j}{x_j^2 + v_j^2} \qquad (7)$$

$$\tau \dot{x}_i = v_i \qquad (8)$$

At this point the coupled oscillators were simulated (see Matlab code in appendix) and the coupling strengths were experimentally determined to be satisfactory in the range [-0.7, 0.7].

2.4 Choice of free and fixed parameters

To summarize, in order to obtain suitable oscillations, different combinations of the parameters τ, α, E, a_{ij} and b_{ij}, of equation (7) must be tried out. One option is to tune all parameters with the optimization algorithms. However, in order to increase the likelihood of convergence it is better to hold down the dimensionality of the search space. Therefore, it is favourable to fix some parameters whilst the most relevant ones are kept free.

As explained in the previous paragraph, the coupling strengths a_{ij} and b_{ij} must be free because the oscillators' synchronization phase depends on them. The oscillation amplitudes controlled by E must also be free because it is not known beforehand how large joint movements should be. Small undulations are required in some modules while larger ones are necessary for others; it depends on the role, for example "hip", "knee" or "ankle" of a particular module in the robot's body.

Oscillations in the region of the module's 0°-angle (the horizontal position in Figure 17) are too restricted. For example in quadruped robots, it is obvious that the "knee" joints should oscillate in a region below the 0°-angle, somewhat downwards, in order to allow the foot to touch the ground and push the body forwards during retraction[2], while staying off the ground during protraction. In fact, the optimal oscillation basis angle of most modules is not the 0°-angle. In general, this angle is not known beforehand and should therefore be another free parameter, that we call here x_0.

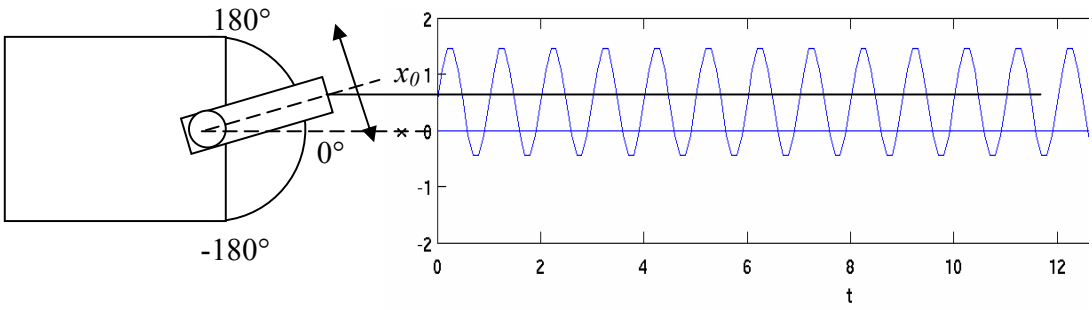


Figure 17: Module's oscillations around x_0.

As discussed before, the oscillation period determined by τ must be common to every oscillator to facilitate synchronization; therefore τ is fixed. The convergence factor α can also be fixed because it influences only the stabilisation period and not the post-synchronization phase.

Parameter	Type	Range/Value	Controls
E	free	$[10^{-8}, (\pi/2)^2]$	Oscillation amplitude
x_0	free	$[-\pi/2, \pi/2]$	Oscillation central angle
a_{ij} and b_{ij}	free	[-0.7, 0.7]	Coupling strength
τ	fixed	$1/(2\pi)$	Oscillation period
α	fixed	0.5	Convergence speed

Table 1: Free and fixed parameters.

The actual number of parameters in a controller depends not only on number of modules but also on the inter-oscillators connections. There are two free parameters (E and x_0) for each module and two (a_{ij} and b_{ij}) for each inter-oscillator connection. The τ and α remain fixed. This is summarized in Table 1 together with the corresponding optimization ranges or fixed values that were used.

2.5 Controller encoding

In order to be independent of the actual optimisation method used, all the above-mentioned parameters are converted to floating-point numbers in the range [0, 1]. A specific controller is simply an array of floating point numbers initialized with random numbers taken from a uniform distribution in this range. Whatever optimization method is used, the parameters are not allowed to exceed the [0, 1] range later on. This restriction limits the diversity of the

[2] Retraction: limb movement towards the rear. Protraction: limb movement towards the front.

results but it also has the advantage of keeping the size of the search space constant. The encoding format is summarized below:

$x_{0,1}$	A_1	a_{11}	b_{11}	a_{12}	b_{12}	...
$x_{0,2}$	A_2	a_{21}	b_{21}	...	...	...
...						
$x_{0,N}$	A_N	a_{N1}	b_{N1}	...	...	...

Table 2: Parameters encoding.

where each row represents the parameters of a single module and the corresponding oscillator. The row length depends on the number of incoming connections. A_i is the module's oscillation amplitude $(A_i=\sqrt{E_i})$. Note that only the genetic algorithm is dependent on the memory arrangement of the parameters because of the crossover operation.

2.6 Coupling

There are many possible ways of coupling the oscillators. Little biological data is available on that topic therefore; it is difficult to prefer a particular solution rather than another one. Intuition suggests that the oscillators must be connected between neighbouring modules, for example from the head down the spine with extensions to the limbs as in [Ijspeert, et al. 2003]. This seems to be the most biologically plausible solution.

Figure 18: Unilateral vs. bilateral coupling.

The choice between unidirectional or bidirectional connections is also difficult because synchronization can work in both situations. Two oscillators can synchronize as long as there is at least one connection between them. In the case of unilateral coupling (Figure 18, left), the frequency of oscillator 2 will shift towards the frequency of the oscillator 1. In bilaterally coupled oscillator (Figure 18, right), the frequency of both oscillators will shift to a value somewhere in the middle of the uncoupled frequencies of both oscillators.

A biologically inspired approach requires bidirectional connections because this allows feedback. For example, if the trajectory of a robot's leg was impeded during its forward swing, with bi-directional connections, the whole robot body could respond.

On the other hand, unidirectional connections allow keeping the number of parameters low. In the end, it was arbitrarily decided use bidirectional coupling for the wheel and caterpillar robots and unidirectional coupling for the tetrapod and quadruped (Figure 19). The solid (black) circles represent the "hip" joint oscillators and the dashed (grey) circles represent the "knee" joints oscillators. Section 3.4 will compare the respective performance of unidirectional and bidirectional coupling in the tetrapod.

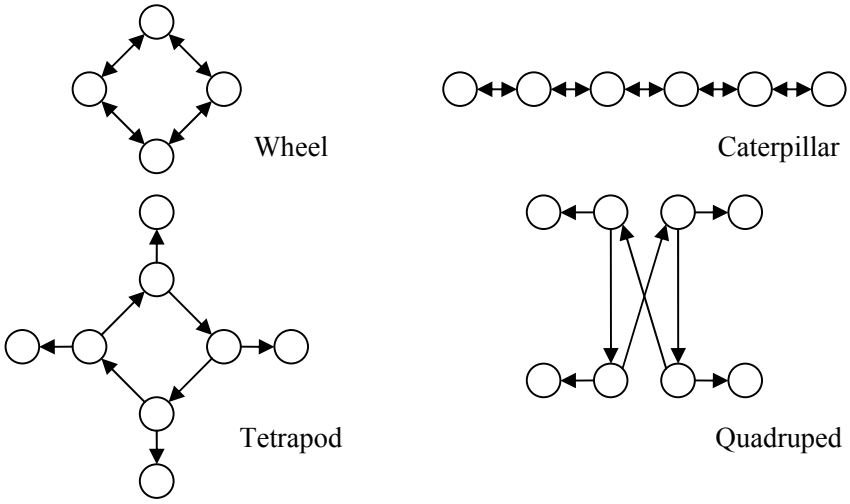


Figure 19: Oscillators coupling.

2.7 Performance measurement

In order to measure its performance[3], every controller is tested during 32 simulated seconds. Throughout this period, a controller should move as far as possible away from its initial location. However, the performance could not be simply measured as the straight distance between the start and end location, because in some cases the robot makes a circle and stops close to where it started. In such cases, the performance evaluates poorly even though, only a tiny parameter change would be required in order to correct the robot's bent trajectory. To overcome this problem, the cumulated or integrated ground distance was also integrated into the performance evaluation.

However, controllers moving in a straight line should still be favoured over zigzagging ones. Therefore, the performance needs to reflect both straight and the integrated distances. For this reason, in this project, the performance was calculated as the weighted sum of both, using this formula:

$$\Phi = \alpha \left\| \vec{p}_N - \vec{p}_1 \right\| + \beta \sum_{i=1}^{N-1} \left\| \vec{p}_{i+1} - \vec{p}_i \right\| \qquad (9)$$

where Φ represents the measured performance, where p_i is the i^{th} point sampled on the robot trajectory, where N is the total number of sampled point, and where α and β are coefficients that allow balancing the respective weights of the absolute and integrated distances. In our simulations these coefficients were set to α=1 and β=1. In order to avoid granting the robots performance scores for plain vibrations, the trajectory points p_i are sampled at 1.0 second intervals such that a robot is always approximately in the same posture when sampling occurs.

Alternatively, another idea was to measure performance as the absolute distance adjusted by a measure of the curvature of the trajectory. The curvature could be computed as the second derivative of the trajectory:

[3] The word *performance* is preferred in this document, because *fitness* makes sense in the context of genetic algorithms but not with the other optimization methods used here.

$$\Phi = \alpha \left\| \vec{p}_N - \vec{p}_0 \right\| + \beta \int \left(\frac{d^2 p}{dt^2} \right) dt \qquad (10)$$

However, this approach was not tried out due to the time limitation.

2.8 Numerical optimization methods

Tuning controllers parameters for locomotion is a multidimensional optimization problem that consists in finding the optimum of a function of several independent variable such that $\Phi = f(p_1, p_2, p_3 \ldots p_N)$ and where Φ must be optimized. Some optimization problems can be solved analytically or by gradient descent methods. However, with many real world problems, these methods do not work because *f(...)* cannot be described by a simple formula. This is the case for the current problem where *f(...)* is computed by the complex algorithms of the physics simulator.

Some methods such as the *downhill simplex* can perform optimization without requiring gradient information. Unfortunately, such non-stochastic methods suffer from their inability to escape from local minima. Therefore, methods involving a random variable are preferred. Three such methods were chosen to optimize our controllers: *genetic algorithms, simulated annealing* and *particles swarm optimization*. In addition, *random search* is used as control technique to see how the other three methods compare with pure chance.

2.8.1 Genetic algorithm

Genetic algorithms are optimization techniques invented by John Holland in the seventies and inspired by the Darwinian theory of evolution. Darwin's theory is based on three basic principles, which are *variation*, *heredity* and *selection*. In nature, these three principles combine into a powerful optimization mechanism that maximizes the chances of survival of genes and potentially explains the existence and the adaptation of complex life forms. The great idea of John Holland was to use the same Darwinian principles in computer simulations in order to solve engineering or mathematical problems.

A genetic algorithm is a kind of artificial evolution of a population throughout a number of simulated generations. The population is made of candidate solutions to a domain specific problem. At every generation, the fitness of the candidates is evaluated with respect to the domain specific objective. The best candidates are selected and allowed to reproduce; this is the *selection* principle. Reproduction generates new offspring based on the genetic material of two parents; this is the *heredity* principle. With a certain probability, the new offspring goes through mutations and therefore differ from its parents; this is the *variation* principle.

Throughout the rest of this document, the genetic algorithm referred to a population of 100 individuals selected by linear rank. From one generation to the next, an elite composed of the 5% fittest individuals is preserved unchanged. From the remaining non-elite 95%, a 5% part is reproduced sexually, using single-point crossover and no mutation. The last remaining 95% are reproduced asexually and mutated. The mutation probability of every parameter is 0.06, which represents roughly two mutations per genome. The mutation amplitude is taken from a normal distribution with mean 0 and standard deviation 0.2. Section 3.5 describes a test in which crossover was disabled in order to compare the performance of sexual and asexual reproduction.

2.8.2 Simulated annealing

Simulated annealing [Kirkpatrick, et al. 1983] is a numerical optimization technique whose name is an analogy with the way in which a metal cools and freezes into a crystalline structure that minimizes its internal energy. At high temperature, the molecules of a liquid move freely with respect to one another. When the liquid is cooled, the molecules' thermal mobility is lost. If the cooling process is carried out slowly enough, the atoms line themselves up into a pure crystal, which is also the minimal energy state of the system. Otherwise, if the cooling is too quick the atoms organize in a less pure structure of higher energy.

With simulated annealing, a virtual cooling process is carried out. The wished final low energy state corresponds to the optimization objective to reach. The molecules' being organized corresponds to the parameters being optimized.

Simulated annealing is a kind of random search with a single individual. At every step, a candidate solution is generated by randomly varying the current solution. Then, this candidate solution is evaluated to find out whether it corresponds to an increase or decrease of the current energy state of the system. If the energy would be decreased then the candidate solution is accepted and it replaces the current solution. If the energy would be increased, then the candidate is usually rejected. However, with some probability, decreasing throughout the cooling process, a candidate can also be accepted if an increase of the energy state results.

This annealing process is illustrated in Figure 20, where parameter changes are shown during a cooling period covering 60 different temperatures. One can observe that at the beginning of the annealing(left), when the temperature is still high, parameter changes take place frequently, whereas towards the end (right), as the temperature cools down, almost no change takes place.

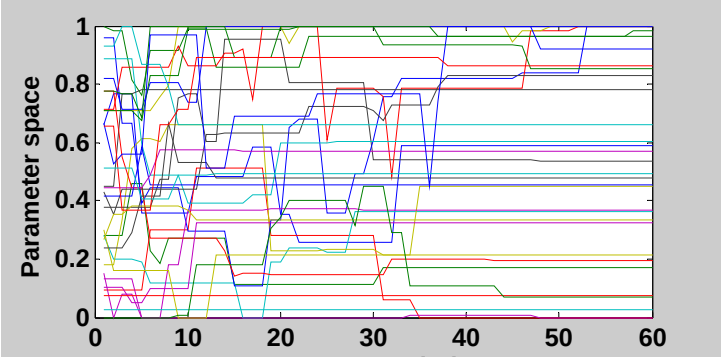


Figure 20: Example of cooling (Tetrapod 32 parameters).

Accepting an increase in energy with a diminishing probability is the main idea behind simulated annealing. When the cooling process terminates, a change that increases the energy level can only be accepted with an infinitesimal probability. Here is the formula:

$$p = \begin{cases} \exp(\frac{-\Delta E}{kT}) & if \quad \Delta E > 0 \\ 1 & if \quad \Delta E < 0 \end{cases} \qquad (11)$$

where p is the probability to accept a candidate solution, T is the temperature which decreases exponentially throughout the cooling process, k is the Boltzmann's constant, ΔE is the energy difference between the candidate and the current solution: $\Delta E = E_{current} - E_{candidate}$.

Simulated annealing works well for combinatorial problems such as the *travelling salesman*. In practice, it is used for example to find the arrangement of integrated circuits that minimizes the interferences between their connecting wires [Kirkpatrick, et al. 1983].

This document refers to an annealing process where the temperature was initialized with 1.0 and was reduced of 0.5 % after every fifth successful parameter change. Configuration changes were generated by the same type of mutation operation as used with the genetic algorithms. Each parameter is mutated with a probability of 0.06 and with amplitude chosen from a normal distribution with mean 0 and standard deviation 0.2.

2.8.3 Particle Swarm optimisation

Particle Swarm Optimisation (PSO) was originally developed in 1995 by James Kennedy and Russell Eberhart [Kennedy, et al. 1995]. Like genetic algorithms, PSO is based on a population that slowly converges towards one or more solutions. However, with PSO, the particles are preserved throughout the entire process; they do not die. Contrary to GA, which is based on competition for better chances of survival and reproduction, PSO uses a kind of cooperation between the particles. This is achieved through the exchange of the coordinates of the best solutions that have been encountered so far.

PSO's particles are simple search agents that "fly" through the search space. Whilst moving, they record the best position that they have discovered so far. They communicate with their neighbours and learn, from them, the best local solution. PSO is based on the concepts of social interaction or more exactly, the tendency of an individual to go his own way, as opposed to his tendency to follow his group's way. At every time step, a particle's flight direction is driven by three factors: first, its own inertial speed, second, its tendency to return to the best solution it has discovered so far, and third, the tendency to go towards the best solution discovered by its neighbours. This can be summarized in equation (12) which calculates the new flight speed of a particle at time t+1, and in equation (13) which calculates the new position a particle:

$$v_{id}(t+1) = \omega\ v_{id}(t) + \varphi_1 rand()(p_{id} - x_{id}(t)) + \varphi_2 rand()(p_{gd} - x_{id}(t)) \qquad (12)$$

$$x_{id}(t+1) = x_{id}(t) + v_{id}(t) \qquad (13)$$

where v is the speed of a particle, i is a particle index, d represents the d^{th} dimension in the parameter space, t is the discrete time index, ω is the particle *speed inertia factor* and where the function *rand*() returns a uniformly distributed random number in the range [0, 1]. The coefficients φ_1 and φ_2 control the individual and social levels of confidence, e.g., how much a particle should follow its own best solution or his group's best solution. Finally p_i is the best previous position of particle i, and p_g is the best previous position in the neighbourhood of particle i. These principles are illustrated in Figure 7 below.

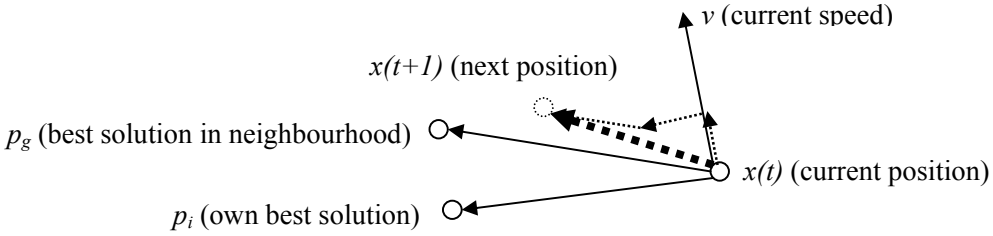


Figure 21: Principle of swarm particle optimization.

In PSO, we speak about the particles' *neighbourhood*. A particle's *neighbourhood* defines from which other particles the information will be received. The neighbourhood size can vary from a few particles to the entire swarm. The neighbourhood type does also vary: some PSO techniques are based on so-called *social neighbourhood*, while others use *geometrical neighbourhood*. In social neighbourhood, the particles are associated with other particles from the beginning and their relationship is maintained throughout the process. A geometrical neighbourhood is defined in accordance with the current particles "proximity" in the parameter space. In this case, the particle-to-particle distances needs to be recomputed at every iteration. The usually mentioned advantage of social neighbourhood is its lower computational burden compared to geometrical neighbourhood. However, in our case, geometrical neighbourhood was preferred, because the processing power required by the optimization algorithm are insignificant anyway compared to that of the physics simulation.

The speed inertia factor ω can either be fixed or decreased during the optimization process. Some authors [Shi, et al. 1998] suggest that a decreasing inertia factor gives better results, and so this approach used here.

In this project, the neighbourhood was based on geometrical proximity with a size of either 50 (the whole swarm) or five particles, according to the experiment. The initial speed inertia was 1.0 and was decreased by 0.5% after every iteration. In every dimension, a particle's position was constrained in the range [0, 1] while its speed was limited to a maximal change of 0.2 per iteration.

2.8.4 Random Search

Random search is straightforward and used only as a method to compare other algorithms. In random search, controllers are generated and initialised with parameters from a random uniform distribution in the range [0, 1]. The generated controllers are tried in sequence. Each time a better controller is found, it replaces the current solution, which is deleted.

2.8.5 Comparison methods

For every optimization algorithm, the controllers' performance was measured using, the same method that was described earlier (Section 2.7). Yet, what is important is to compare the algorithms performance with each other, on a common scale representing the processor time. However, the various algorithms are based on different time bases: for example *generations* for GAs, *annealing schedule* for SA and *iterations* for PSO. As a result, every algorithm was implemented such as to terminate and deliver results after 12,000 controller evaluations and a time scale that represents the number of such evaluations between 0 and 12,000, was used.

2.9 Simulation

Clearly, chaotic or unsynchronized oscillations tend to result in inefficient locomotion. In this project, the task of rejecting defective solutions was left entirely to the optimisation algorithms. No preliminary test, of the oscillations' "quality" was performed before submitting a controller to the simulator. One can argue that this approach might slow down the optimization processes because many non-viable solutions are expensively simulated. However, it is also possible that poorly synchronized oscillators are very close to good solutions in parameter space. In which case, their premature removal would also be detrimental.

2.9.1 Simulation parameters

Within Webots, each controller was evaluated during 32 simulated seconds, which corresponds roughly to one second of processor time, depending on the complexity of the model. In the configuration used, new servomotor positions were transferred from the robot controller to the simulator every 64 milliseconds. The physics integration step was of 32 milliseconds. The oscillators used Euler integration with a step of one millisecond. The ODE Coulomb friction was set to 0.9 for all robot modules and in every experiment.

2.9.2 Noise

The search algorithms used here are stochastic and as a result, obviously different results are obtained for every search. In the contrary, the behaviour of the oscillators is completely deterministic; if the same initial conditions and parameters are used, the controller generates the same sequence of servomotor commands every time. However, some noise was present in the simulations, produced by the simulator's underlying physics engine (ODE). Consequently, the measured performance of a controller varied. This is illustrated in Figure 22, which shows the performance of the four morphologies, tested 20 times with the same controller.

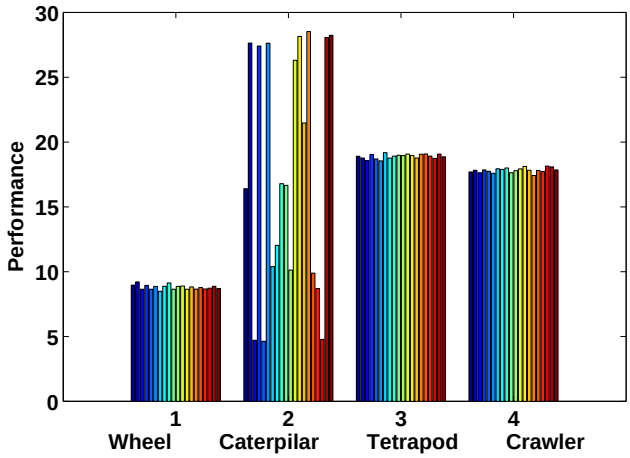


Figure 22: Discrepancy in performance due to noise.

In this example, the performance divergence was a lot larger in the caterpillar than in the other morphologies. The reason for this is that the caterpillar robot sometimes falls over and, once on the side, it cannot move any further. In fact, falling over does also sometimes occurs to the wheel robot, but in this example, a robust controller was used and therefore, this did not

happen. Actually, the wheel robot is actually very sensitive to noise because is moves by a kind of vibrations, not using a proper gait like the others robots; therefore, its performance usually varies a lot.

Noise, in the performance measurement, delays the algorithms convergence. However, a certain amount of noise is representative of the real world, e.g. inaccuracy in the servomotors or irregular ground friction. Therefore, noise improves the robustness of the results and the chances that the simulation results can be transferred to a real robot.

2.9.3 Numerical instabilities

In addition to the noise, numerical instabilities appeared in the simulations. Some of these instabilities originated from the oscillators. They could be detected automatically in software; when the current state of an oscillator exceeded a specific security threshold or equalled Not-a-Number (NaN), an arbitrary performance of zero was assigned to the corresponding controller.

Furthermore, sometimes a controller reported an outstanding result that could unfortunately not be reproduced afterwards. This type of situation was due to numerical instabilities in ODE. To overcome this kind of problem, the usual solution is to decrease the integration step of the physics simulator. However, a smaller integration steps considerably slows down ODE and therefore the whole search. Therefore, we rather chose to recheck all the results manually and to redo the searches that gave irreproducible results.

3 Results

A series of 100 searches, representing around 300 hours of processor time in total, was carried out. The four described morphologies were tested with the different search methods, further denoted: GA for genetic algorithms, SA for simulated annealing, PSO5 for particle swarm optimization using a geometrical neighbourhood of five particles, PSO50 for particle swarm optimization with a "whole swarm" neighbourhood and finally RS for random search. In addition, each test was repeated five times for averaging the results.

3.1 Main results

In Figure 23 below, the learning curves of the search algorithms for all four morphologies are plotted. Each curve represents the current best solution of the corresponding algorithm throughout a search and it is computed as the average of five different searches.

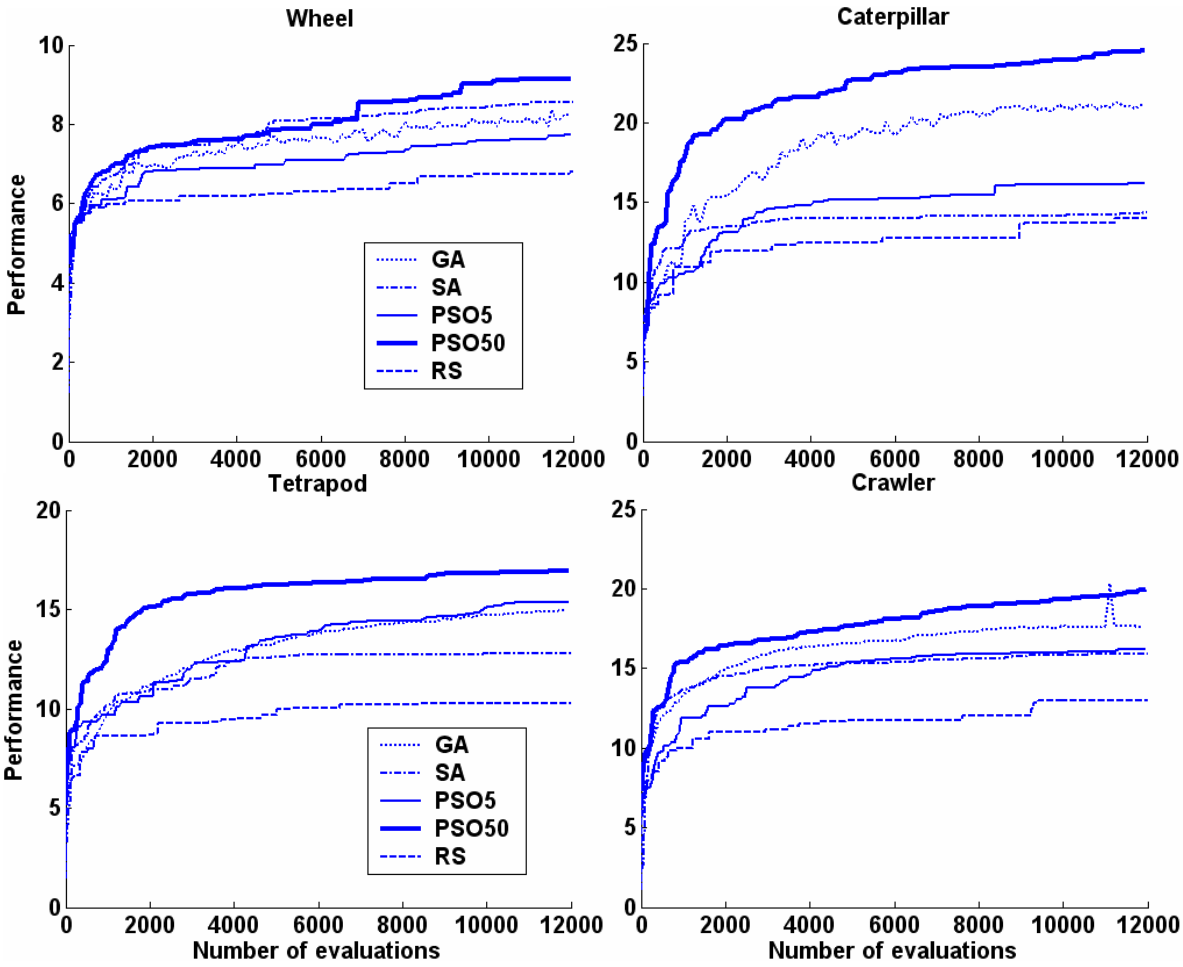


Figure 23: Compared learning curves of five search methods for each morphology (results are averaged over five searches).

The horizontal axes indicate the number of controller evaluation between 0 and 12,000, which corresponds also to 120 generations for GA or 240 iterations for PSO. What is important to notice is that for any morphology, the best results were always obtained using PSO50. The

worst results were constantly obtained by random search, as expected. The small peak in the GA curve in the fourth diagram is due to a numerical instability, which disappeared in the subsequent generation, because the genotype evaluated then to a different performance.

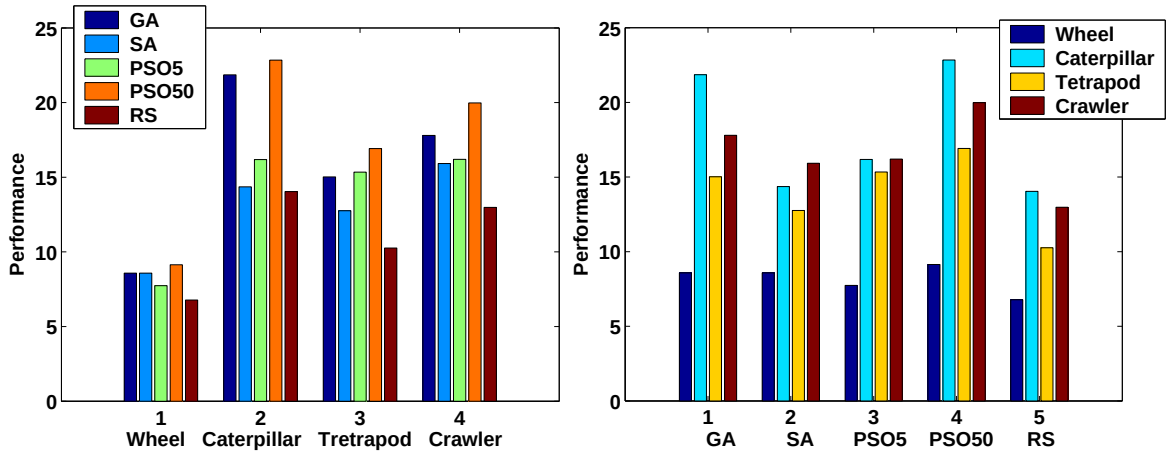


Figure 24: Performance results, sorted by morphology (left) and algorithm (right).

Figure 24 summarizes these results in another form: only the final best performances are compared. On the left, the results are grouped by morphology and on the right, by algorithm. Without considering the search methods, the caterpillar is usually the fastest morphology but is sometimes outperformed or equalled by the crawler.

3.2 Gait description

This paragraph will shortly go through the description of some of the most efficient gaits obtained. Video clips of precisely the same results that are described here are available for closer examination on the webpage: http://www.yvanbourquin.com/ModularWalkers.

3.2.1 Wheel

This sequence was obtained with the genetic algorithm; it represents a full "gait" cycle of the wheel robot. The interval between two images is 250 milliseconds. The wheel moves through vibrations while standing on two limbs. In most solutions, the upper rear limb is rather immobile and used as a swinging mass.

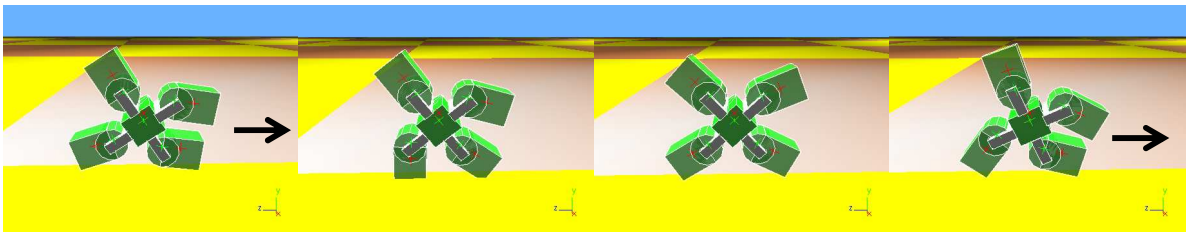


Figure 25: Example of wheel vibrating "gait".

The wheel did not develop a rotational motion, as was expected. The reason is probably that the oscillator's frequency is fixed to 1 Hz, and 1 second is not enough for the motors to carry out a full rotation of the structure.

3.2.2 Caterpillar

The results of the 25 best caterpillar controllers can be classified in three categories (Table 3). A first category corresponds to controllers that move as a kind of “rolling loop” similar to the caterpillars of tracked vehicles (Figure 26). A second category corresponds to controllers producing caterpillar-like motion, similar to real worms or caterpillars. The third category is made of robots moving by vibrations. Two subcategories a) and b) were also distinguished, whether to a controller chose to move in "head" or "tail" direction.

	"Gait" category	**Number of results**	**Mean performance**
1	Rolling loop, a) tail first, b) head first	7 (2 + 5)	25.0
2	Caterpillar gait, a) tail first, b) head first	12 (9 + 3)	16.1
3	Motion by vibration	6	10.5

Table 3: Caterpillar "gait" categories.

Note that, with an average performance of 25.0, the first category, the rolling loop, is by far the fastest caterpillar, and actually the overall best performance of all simulations. Another thing to notice is that categories 1 and 2 seem to have a preferred direction. This can be understood because the caterpillar robot model is not front-rear symmetrical. Unfortunately, the travelled distance is measured in the centre of the robot's tail and therefore, gaits showing large tail movements were artificially favoured.

Five results from each optimization method are not enough to make a conclusion whether one optimization method is more likely to converge towards a particular category, but there is no reason to think that this would be the case.

Figure 26: Example of caterpillar's rolling "gait".

In every rolling loop result (Figure 26), the caterpillar robots had a "gait" cycle lasting two seconds, therefore exactly twice duration of one period of oscillations. This surprising result is in fact the only possible outcome, because only two similar bending waves propagated along the body can provoke a full rotation. In Figure 26, the upper row of images corresponds to the first travelling wave, and the lower row of images corresponds to the second travelling wave, with the same motor activation angles.

This rolling loop “gait” can be understood more easily by looking at the corresponding oscillations. In Figure 27, a small and regular phase shift, indicated by the oblique line, is propagated during one second from module 2 to module 6.

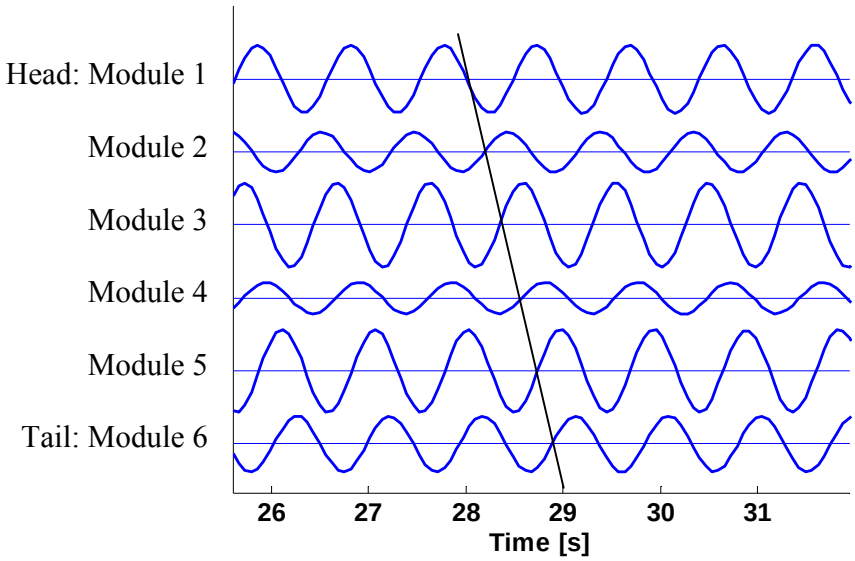


Figure 27: Example of caterpillar oscillations.

In this example the "head" module activation does not take part to this travelling wave; it is actually in counter phase. The reason is that the head's hinge is not connected to another module, and therefore it behaviour is rather irrelevant in this type of locomotion.

3.2.3 Tetrapod

With the tetrapod morphology, the motors have enough power to lift the body above the ground during the whole gait cycle. Therefore, an upright posture (Figure 28, left) emerged even though in the initial position the robot is "sitting" on the ground. Independently of the algorithm used, most of the time the controllers developed a kind of "walk" similar to that found in animal quadrupeds.

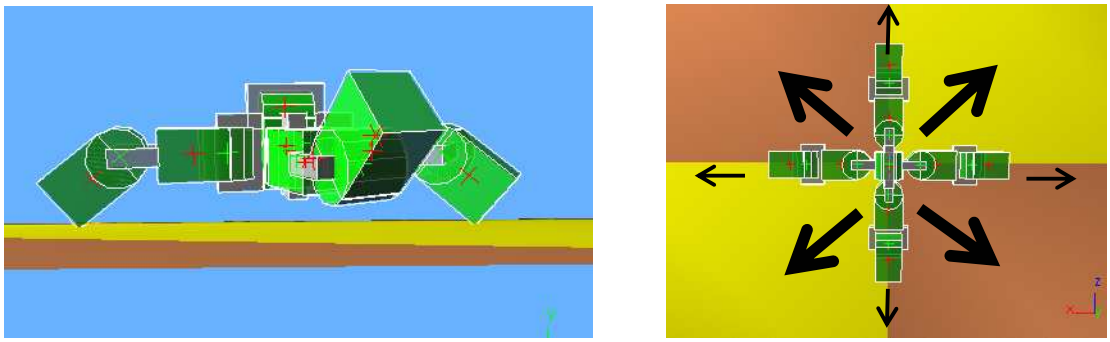


Figure 28: Upright tetrapod and axes of symmetry.

Usually a kind of 2-phases gait appeared in which two diagonally opposed legs push simultaneously and propel the body while the two other legs swing forwards. However, these movements are not perfectly in phase and can therefore be considered as 4-phase gaits.

Even though the tetrapod is 8-ways symmetrical (Figure 28, right), almost every examined solution evolved to walk diagonally with respect to the central module (large arrows). They use two legs on each side, like actual quadrupeds. From 25 examined results, only one evolved to take advantage of other axes of symmetry (small arrows). In that result, the front and hind legs remained immobile and the two side legs pushed in phase.

This sequence shown in Figure 29 was obtained with PSO50. It is very representative of the results. Three legs are used efficiently; the feet describe roughly elliptic trajectories in which the "knee" joints are bent downwards during the retraction and bent upwards during protraction in a way similar to that of many vertebrates.

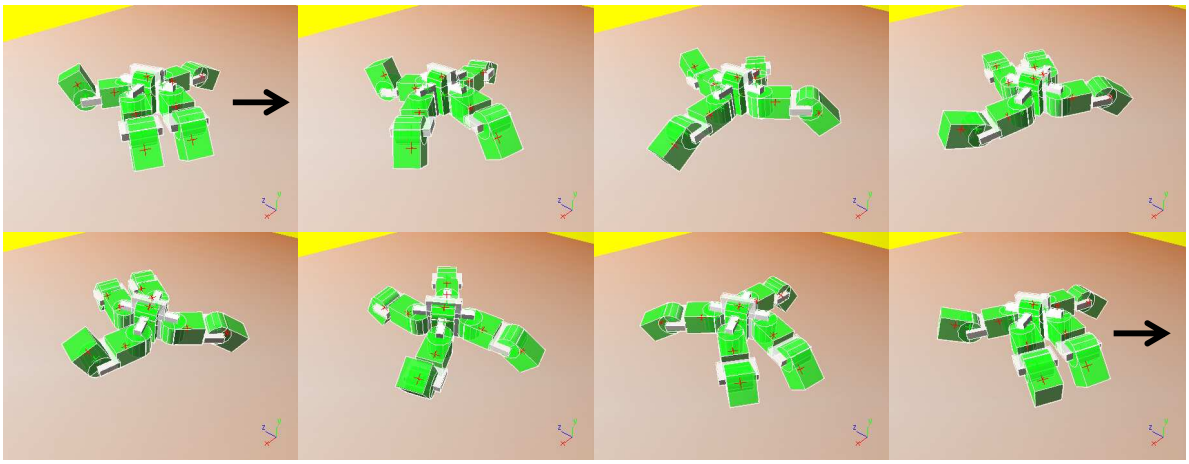


Figure 29: Example of tetrapod gait (oblique view).

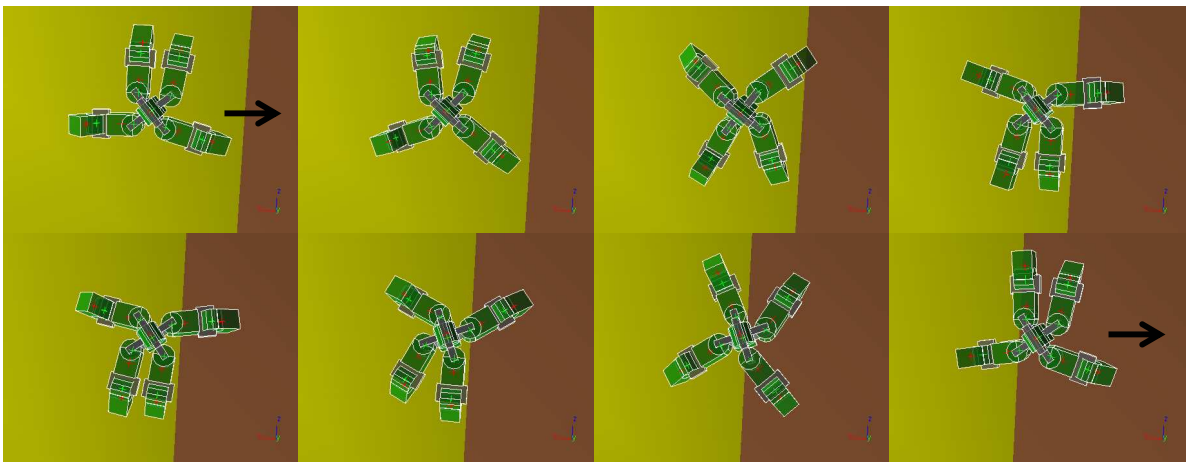


Figure 30: Example of tetrapod gait (top view).

In Figure 31, the same result is shown in the form of oscillations. In the "hip" motors, a positive value corresponds to a forwards limbs orientation and a negative value corresponds to backwards. We can see that crosswise-opposed legs are in phase: the front left[4] hip motor (1) is roughly in phase with the hind right hip motor (4). Similarly, motor (2) is roughly in phase with motor (3).

[4] Since the tetrapod is actually a symmetrical morphology, the terms *left*, *right*, *front* and *hind* must here to be understood with respect to the actual direction of motion.

However, the "knee" oscillations seem less efficient. In the "knee" motors, positive values mean upwards and negative values means downwards, null is the horizontal. An efficient quadruped gait corresponds to the knee pointing downwards during retraction and upwards during protraction. In our example, this corresponds to a counter-phase synchronization of each "knee" with the corresponding "hip".

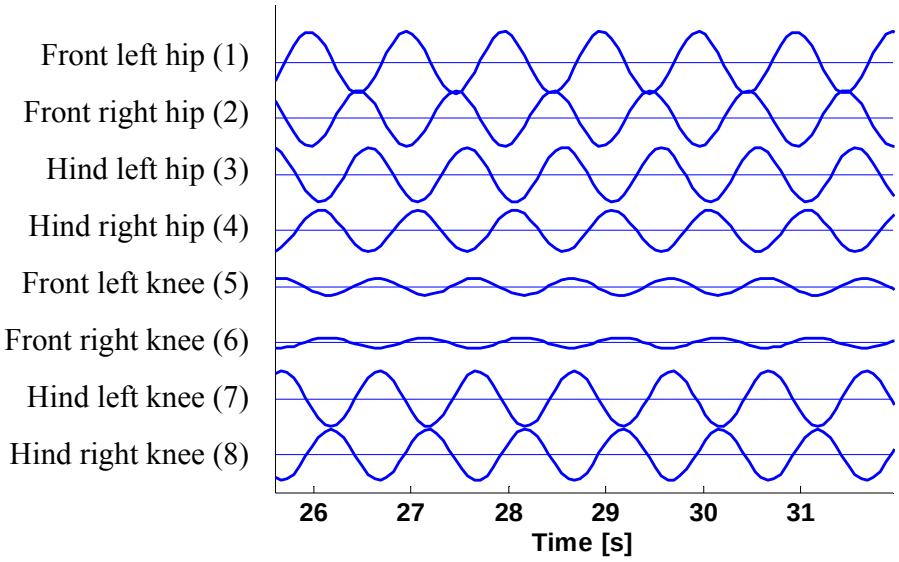
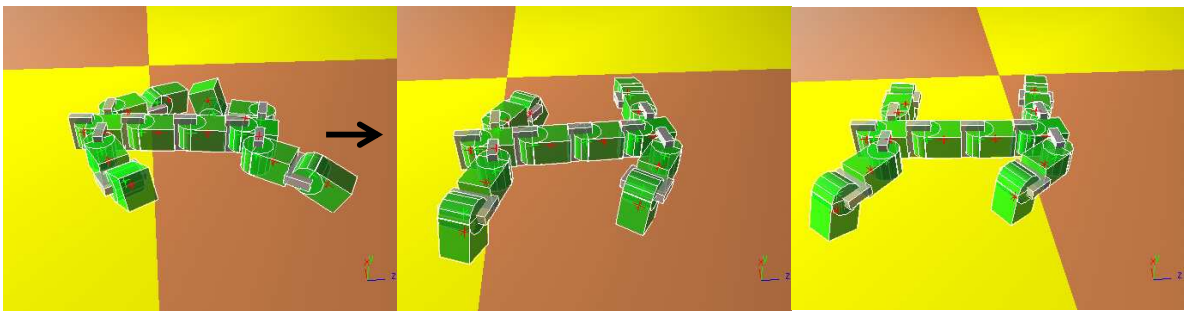


Figure 31: Example of tetrapod oscillations.

Here, the solution is imperfect; one can observe that there is an approximate counter-phase, but unfortunately only between the hind hips and the corresponding hind knees. The two front legs are used inefficiently because they points downwards during protraction. However, the oscillation amplitude is small and therefore this is apparently not hindering the robot too much.

3.2.4 Crawler

It is hard to classify the crawler results, because they differ only by slight variations. All the results use three or four legs. As with the tetrapod above, more or less efficient gaits were found in which, usually, the “knee” joints point downwards during retraction and more or less horizontally or upwards during protraction. The next sequence represents a typical crawler result. It was optimized by simulated annealing. These images are taken at 160 milliseconds intervals.



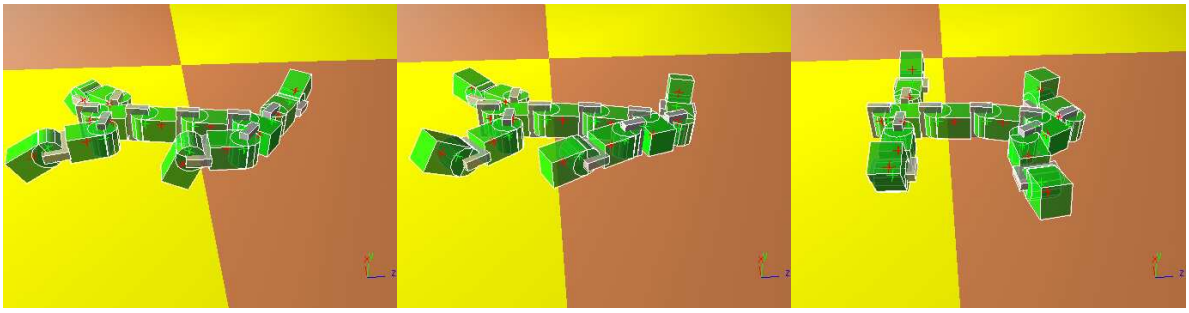


Figure 32: Example of crawler gait.

The gait shown here has 2-phases but is asymmetrical (Figure 32). The first row of images correspond to retraction, the second one corresponds to protraction. Three of the legs move in phase while the fourth one (left front leg), moves in counter phase. The crawler's body lies on ground during the protraction but is lifted slightly during the retraction, much like a reptile's sprawling posture. This is different from the tetrapod, which is lighter and keeps its body off the ground during the complete cycle.

Note that from 25 results, 10 chose to start moving in the direction shown in Figure 32 and the 15 others started in the opposite direction. There is no known advantage to either solution. Sometimes, the robots moved slightly sideways, not exactly parallel to the robot's spine.

3.3 Genotype analysis

At this point, it is interesting to compare the results of the various algorithms to see if there is convergence to a common solution. Figure 33 shows the 25 best crawler controllers obtained with various search methods. The thicker line shows the mean value of all the controllers. There is rather large diversity in the solutions and only a slight tendency towards a particular configuration.

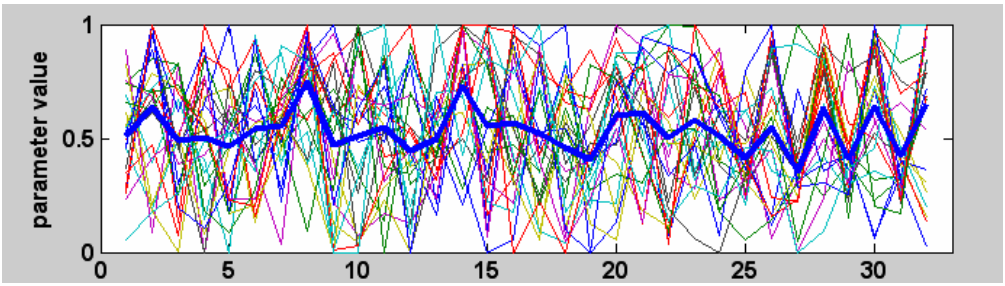


Figure 33: Parameters sets of the 25 best crawler controllers (the x-axis corresponds to the parameter numbers).

When examining the standard deviation of these parameters (Figure 34), one can notice that the smallest variance correspond quite precisely to the controller's x_0 parameters (red bars), while the largest variance correspond to the oscillations amplitudes A (yellow bars) of the controllers. The x_0 parameters correspond to the initial oscillation angles of the "knee" (parameters 17, 21, 25 and 29) and "hip" joints (parameters 1, 5, 9 and 13). Therefore, in the "knee" joints, this consistency of x_0 indicates the standard downwards orientation of the "tibia" required to ensure contact with the ground.

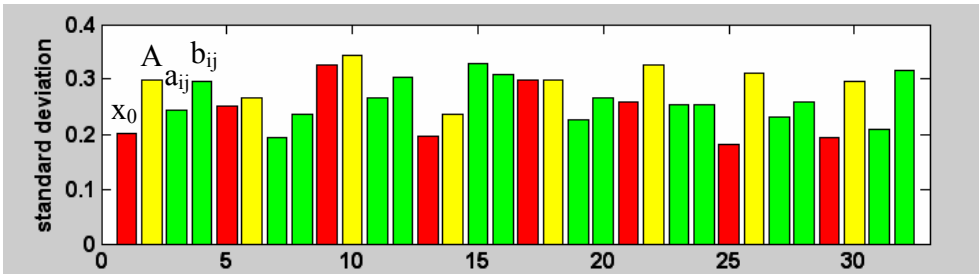


Figure 34: Standard deviation of the 25 best crawler controllers (the x-axis corresponds to parameter numbers).

In the "hip" joints, this corresponds to "femur" angles usually oriented perpendicularly to the spine, as is required to enable larger pushing movements. Clearly, the small variation in the x_0 parameters reflects the general posture required for the locomotion of the crawler. On the contrary, the larger variance in the a_{ij} and b_{ij} parameters (green bars) reflects the fact that several different combinations of coupling strengths can result in the same phase shifts.

A similar variance scheme was found in the tetrapod morphology. This result is obvious because it is also a quadruped. The reason why the oscillations' amplitudes A are the most variable parameters is not clear. On the contrary, concerning the caterpillar, the most stable parameters correspond to the amplitude and not to the x_0.

3.4 Unidirectional vs. bidirectional connections

In this test, the difference between unidirectional and bidirectional connections was tested using the tetrapod morphology and the genetic algorithm. The same GA parameters as specified in paragraph 2.8.1 are used. The only thing that was changed was the oscillators' connections scheme.

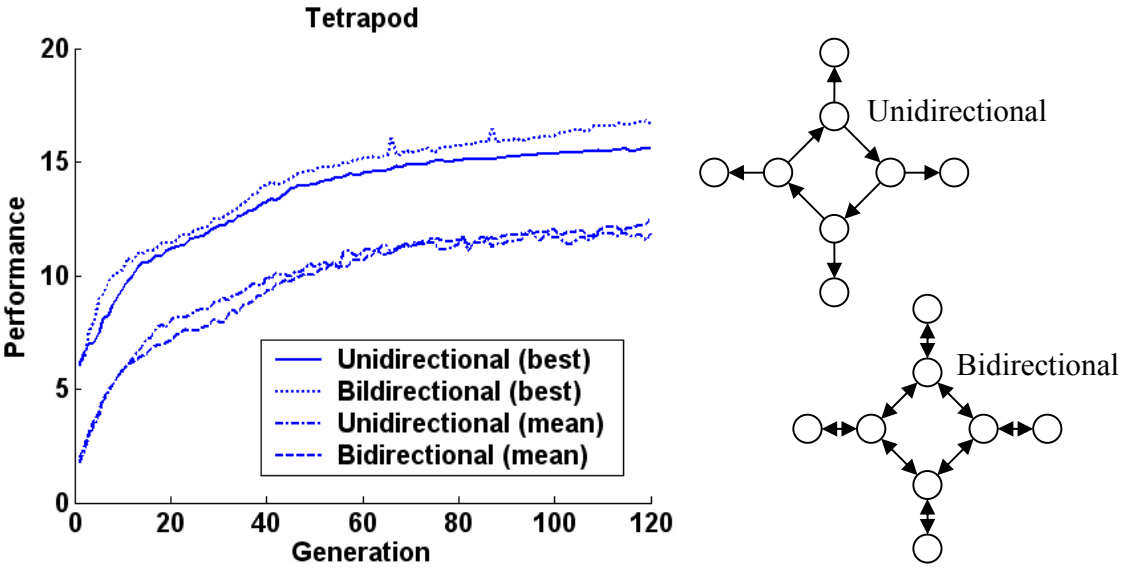


Figure 35: Compared performance of unidirectional and bidirectional tetrapods.

The two upper curves represent the evolution of the best individuals of a population of unidirectional (solid line) and bidirectional tetrapod (dotted line). The two lower lines correspond to the mean individuals. Each curve is averaged over eight different runs of the

genetic algorithm. The results are slightly better for the bidirectional tetrapod but the difference is not significant.

3.5 Sexual vs. asexual genetic algorithm

In this test, the objective was to compare the performance of sexual vs. asexual genetic algorithm. Sixteen GA optimizations were carried out, eight with crossover and eight without crossover. The results were averaged; they are shown in Figure 36. The two upper curves represent the best individuals of the sexual and asexual populations, and the two lower curves represent the corresponding mean individuals. The learning curves are almost identical for sexual and asexual.

We can deduce that the crossover operation employed here is inefficient. Apparently, the sequence of parameters described above (section 2.5) is not constituted of reusable building blocks. In fact, when a single coupling strength is modified in a controller, this can result in completely different phase shifts of the oscillators. Consequently, the crossover will usually produce the same effect as a large mutation resulting in an offspring which performance is unrelated to the one of its parents.

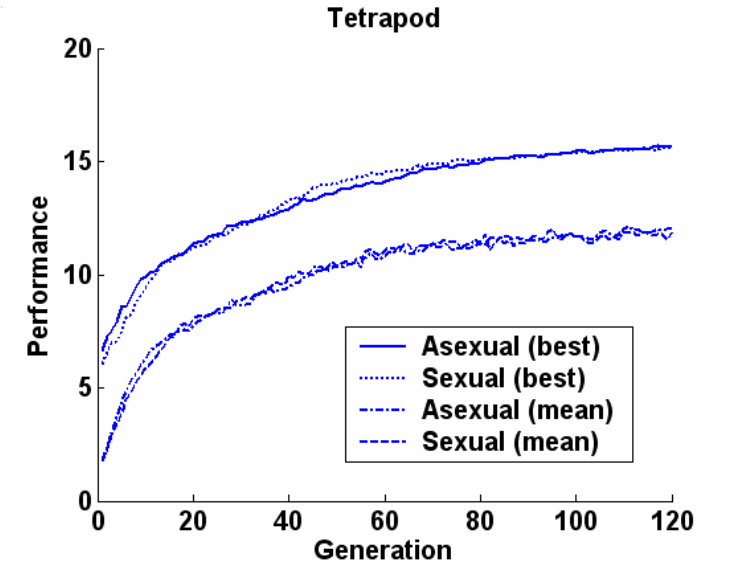


Figure 36: Sexual vs. asexual GA.

Still, it is possible that a crossover operation designed in a different way performs better. The version used here did not consider any building blocks and therefore the crossover points were set completely arbitrarily between any of the floating-point parameters. For example, it might be an improvement to use a crossover operation that does not split adjacent a_{ij} and b_{ij} and adjacent A and x_o, because these parameters are closely related.

4 Discussion

4.1 Algorithms comparison

All these search methods have a lot in common. They start with a set of randomly generated candidate solutions, and then new candidates are generated by varying the current ones. The new candidates' performances are evaluated and then, with some probability better solutions replace the previous ones. What differs is the size of the population, the way the variation is generated and the way replacement is carried out.

In simulated annealing and random search, the set size is one. On the contrary, in genetic algorithm and particle swarm optimization the set is a whole *population*. A population of individuals is generally superior to a single individual because this allows generating variation from several suitable starting points. With a population, the search is pursued in several directions; this explains why GA and PSO constantly delivered the best results.

Furthermore, there are many methods to produce variation. For GA and SA, single point mutations with a certain probability and with an amplitude chosen from a normal distribution were used. However, it would also have been possible to choose the mutation in a hypersphere around the current solutions as in [Slocum, et al. 2000]. Similarly, there are various ways to make use of random in PSO, but it is impossible to try them all. The annoying problem is that, most of the time, mutations give an uphill move that decrease performance. In [Press, et al. 1988], a method is proposed that uses simulated annealing together with the downhill simplex method. This technique can increase the chance of a downhill mutation. In the same way, PSO "flight" speed plays the role of a momentum that potentially increases the chances of a downhill move. This is similar to the momentum used in neural networks to improve the speed of convergence by taking into account the current learning rate. In this aspect, PSO seems superior to GA because it does not simply generates variation from good past solutions, but it also takes into account the current improvement direction. Therefore, it increases the chances that the next mutation will be downhill.

Finally, it is worth noting that not only can the search algorithms can be infinitely varied, but also furthermore, they can be used in combination. For example, both genetic algorithm and particle swarm optimization are quite efficient alone, but eventually they could be more efficient when used together. Since they are both based on a population, it is possible to alternate the technique within the same search. Furthermore, for some GAs the mutation rate is too restrictive for the initial search but fine for the parameters fine-tuning. Therefore, an alternative is to start with a random search and finish the parameters fine-tuning with a GA or any of the other techniques described here.

4.2 Conclusions

Interestingly, the four search methods all produced similar and good results. The biggest surprise was that particle swarm optimization delivered constantly better results than genetic algorithms. The myth that genetic algorithms is a superior optimization technique is a bit broken. In fact, genetic algorithms are not superior but their popularity in evolutionary robotics is mostly due to their nature, inspired from phylogeny, which fits very well with the concept of artificial life or artificial intelligence. This project has shown that PSO works just as well, or even better than GAs, to optimize controllers.

However, we should also avoid generalising the results from this experiment. GA, SA and PSO all use a number of control parameters, e.g. population size, swarm size, mutation rate and so on. These control parameters have a great influence on the results. In this project, the control parameters of the algorithms have been chosen intuitively by the author, but in fact, they would need to be optimized themselves.

Generally, locomotion developed beyond expectations. In the tetrapod and crawler, gaits similar to natural quadrupeds evolved and their performance is apparently close to the theoretical maximum. Although our results are good, when looking at the gaits obtained, one always has the impression that something can still be improved. It is possible that classical engineering methods, for example gait tables or predefined joint trajectories, reaches similar or better performance. It would be interested to compare the results of these two divergent methods.

4.3 Future work

The effectiveness of the generated locomotion gaits has not been verified in hardware. The logical continuation of the project is to transfer the simulation results into the real robot and carry out the calculation of the oscillations on-board. However, at the time of writing, the hardware was not completely ready and therefore this could not be tried out.

An interesting extension to the project would be to implement the control of direction in the tetrapod and crawler robots. If the left and right side CPGs are independent, and if the oscillations' amplitude on one side is changed using the tonic input [Ijspeert, et al. 2000], then, the tetrapod and crawler robots could turn.

Our performance measurement involved only locomotion speed; however, in order to gain in autonomy, robots should also be energy efficient, just as their biological counterpart. This is especially critical for modular robots because, according to their creators' claims, they are aimed at working outdoors. Therefore, the energy consumption should be incorporated in the performance measurement. However, this calculation is not necessarily easy because, a body has is own dynamics. During a gait, the energy can be temporary stored in elastic or gravitational form and restored later on.

Finally, the model used here was very limited compared to a real biological system. Only CPGs were modelled whereas animal locomotion is also controlled by the brain and by sensory feedback. Even though no sensors were implemented in the real robot, it would be interesting to include sensory perception in the simulation. One possibility is to simulate touch sensors in order to know when a foot is in contact with the ground. Another possibility is to implement proprioception and the feedback of the actual motor angles into the oscillators.

5 References

R. D. Beer (1996). *Toward the evolution of dynamical neural networks for minimally cognitive behavior*. Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (pp. 421-429). MIT Press.

R. A. Brooks, A. M. Flynn (1989). *Fast, Cheap and Out of Control: A Robot Invasion of the Solar System*. Journal of the British Interplanetary Society, Vol. 42, pp 478-485, 1989.

D. Floreano, F. Mondada (1996). Evolution *of Plastic Neurocontrollers for Situated Agents.* Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA: MIT Press.

J. C. Gallagher, R. D. Beer, K. S. Espenschied, R. D. Quinn (1996). *Application of evolved locomotion controllers to a hexapod robot*. Robotics and Autonomous Systems 19:95-103.

S. Grillner (1985). *Neurobiological Bases of Rhythmic Motor Acts in Vertebrates*. Science, New Series, Vol. 228, No. 4696 (Apr. 12, 1985), 143-149.

I. Harvey, P. Husbands, D. Cliff, A. Thompson, N. Jakobi (1997). *Evolutionary Robotics: the Sussex Approach*. In Robotics and Autonomous Systems, v. 20 (1997) pp. 205--224.

P. Husbands, T. M. C. Smith, M. O'Shea, N. Jakobi, J. Anderson, A. Philippides (1998). *Brains, Gases and Robots*. In Niklasson, L., Boden, M. and Ziemke, T., editors, Proceedings of the 8th International Conference on Artificial Neural Networks: ICANN98, pages 51-64.

A. J. Ijspeert, M. Arbib (2000). *Visual tracking in simulated salamander locomotion*. Proceedings of the Sixth International Conference on the Simulation of Adaptive Behavior, MIT Press, 2000. pp 88-97.

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

A. J. Ijspeert (2002). *Vertebrate locomotion*. In the Handbook of Brain Theory and Neural Networks, Second Edition, M.A. Arbib (Ed.), Cambridge, MA: The MIT Press, 2002. pp 649-654. The MIT Press.

A. J. Ijspeert, J.-M. Cabelguen (2003). *Gait transition from swimming to walking: investigation of salamander locomotion control using non-linear oscillators*. In Proceedings of Adaptive Motion in Animals and Machines, 2003.

N. Jakobi (1998). *Running Across the Reality Gap: Octopod Locomotion Evolved in Minimal Simulation*. In P. Husbands and J-A Meyer, editors, Proceedings of Evorobot. Springer Verlag, 1998.

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

J. Kennedy, R. Eberhart (1995). *Particle Swarm Optimization*. Proceedings of the 1995 IEEE International Conference on Neural Networks, pp. 1942-1948, IEEE Press.

S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi (1983). *Optimization by Simulated Annealing*. Science, vol. 220, No. 4598, pp. 671-680.

H. Kimura, K. Sakurama, S. Akiyama (1998). *Dynamic Walking and Running of the Quadruped Using Neural Oscillator*. Proceeding of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS1998), Victoria, pp.50-57, 1998.

O. Michel (2004). *Webots: Professional Mobile Robot Simulation.* International Journal of Advanced Robotic Systems, Volume 1 Number 1 (2004), pp 39-42.

B. Mesot (2004). *Self-Organization of Locomotion in Modular Robots*. Unpublished Diploma Thesis. http://birg.epfl.ch/page42735.html

S. Mojon (2004). *Using nonlinear oscillators to control the locomotion of a simulated biped robot*. Unpublished Diploma Thesis. http://birg.epfl.ch/page44565.html

F. Mondada, D. Floreano (1995). *Evolution of neural control structures: Some experiments on mobile robots*. Robotics and Autonomous Systems, 16, 183-195.

A. Pikovsky, M. Rosenblum and J. Kurths (2001). *Synchronization, a universal concept in nonlinear sciences*. Cambridge Nonlinear Sciences Series 12.

W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling (1988). *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, on-line book: http://www.library.cornell.edu/nr/cbookcpdf.html

K. Sims (1994). *Evolving Virtual Creatures*. In SIGGRAPH '94 Proceedings, pages 15-22, July 1994.

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

Y. Shi and R. C. Eberhart (1998). *Parameter selection in particle swarm optimization*. In *Evolutionary Programming VII: Proc. EP98*, New York: Springer-Verlag, pp. 591-600.

M. L. Shik, F. V. Severin, G. N. Orlovskii (1966). *Control of Walking and Running by Means of Electrical Stimulation of the Mid-Brain*. Biophysics, 11:756-765, 1966.

A. C. Slocum, D. C. Downey, R. D. Beer (2000). *Further Experiments in the Evolution of Minimally Cognitive Behavior: From Perceiving Affordances to Selective Attention*. Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior. Cambridge, MA: MIT Press.

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

G. Taga (1994). *Emergence of bipedal locomotion through entrainment among the neuro-musculo-skeletal system and the environment*. Physica D: Nonlinear Phenomena, 75(1-3):190-208, 1994.

E. Yoshida, S. Murata, A. Kamimura, K. Tomita, H. Kurokawa, S. Kokaji (2003). *Evolutionary Synthesis of Dynamic Motion and Reconfiguration Process for Modular Robot M-TRAN*. Proceeding 2003 IEEE International Symposium on Computational Intelligence in Robotics and Automation, July 16-20, 2003, Kobe, Japan.

6 Appendix

6.1 Matlab listings

6.1.1 Limit cycle

```
clear all;
a=0.7;
tau=10;
E=1;
T=500;
hold on;

for i=1:30
    v(1)=rand*4-2;
    x(1)=rand*4-2;
    for t=1:T
        v(t+1)=v(t)+(-a*((x(t)*x(t)+v(t)*v(t)-E)/E)*v(t)-x(t))/tau;
        x(t+1)=x(t)+v(t)/tau;
    end;
    plot(x,v);
end;

axis([-2 2 -2 2]);
set(gca,'FontSize',12);
set(gca,'FontWeight','bold');
xlabel('x');
ylabel('v');
```

6.1.2 Oscillators coupling

```
clear all;

alpha=0.5;     % convergence factor
f=1;           % oscillation frequency [Hz]
tau=f/(2*pi);  % time constant [s]
N=4;           % num oscillators
dT=0.0001;     % integration step [s]
D=32;          % duration of measurement [s]
T=1000;        % num of samples
K=D/dT/T;      % num dT per sample
MINMAX=0.7;    % connection min max

% random oscillation amplitudes
E=(rand(N,1)*(pi/2-0.0001)+0.0001).^2;

%  random initial conditions
v=rand(1,N)*2-1;
x=rand(1,N)*2-1;

% random coupling strengths
a=zeros(N,N);
b=zeros(N,N);
connections=[0 1; 1 0; 1 2; 2 1; 2 3; 3 2; 3 0; 0 3];
for i=1:size(connections,1)
```

```
    a(connections(i,2)+1,connections(i,1)+1)=rand*2*MINMAX-MINMAX;
    b(connections(i,2)+1,connections(i,1)+1)=rand*2*MINMAX-MINMAX;
end;

for t=1:T           % for every measure sample
    for k=1:K       % for every integration steps
        for i=1:N   % for every oscillators
            S=0;
            for j=1:N
                % sum of coupling inputs
                S=S+(a(i,j)*x(j)+b(i,j)*v(j))/(x(j)^2+v(j)^2);
            end;
            % main oscillator equations
            v(i)=v(i)+dT*((-alpha*((x(i)^2+v(i)^2-E(i))/E(i))*v(i)-x(i)+S)/tau);
            x(i)=x(i)+dT*v(i)/tau;
        end;
    end;
    X(:,t)=x(:);
end;

% plot results
for j=1:N
    subplot(N,1,j);
    plot([1:T]*K*dT, X(j,:));
    axis([0, D, -3, 3]);
    xlabel('Time [s]');
    ylabel('x');
end;
```

6.2 Example VRML listings: Tetrapod

```
#VRML_SIM V4.0 utf8
#000000
WorldInfo {
  info [
    "tetrapod walker"
    "yvan bourquin <mail@yvanbourquin.com>"
    "Date: 23 jun 2004"
  ]
  title "modular walker"
  ERP 0.8
}

...

DEF WALKER CustomRobot {
  rotation 0 0 1 0
  children [
    GPS {
      name "gps"
    }
    DEF emitter Emitter {
      name "emitter"
      range 999
    }
    DEF QUAD_GROUP Group {
      children [
```

```
        Transform {
          rotation 1 0 0 1.5708
          children [
            DEF GREEN_BOX_SHAPE Shape {
              appearance DEF PLASTIC Appearance {
                material Material {
                  diffuseColor 0 1 0
                }
              }
              geometry Box {
                size 0.05 0.045 0.062
              }
            }
          ]
        }
        Transform {
          translation 0 0.031 0
          rotation 1 0 0 1.5708
          children [
            DEF GREEN_CYLINDER_SHAPE Shape {
              appearance USE PLASTIC
              geometry Cylinder {
                height 0.045
                radius 0.025
              }
            }
          ]
        }
        Transform {
          translation 0.045 0 0
          rotation 0 1 0 1.5708
          children [
            DEF GREY_BOX_SHAPE Shape {
              appearance DEF ALU_APPEARANCE Appearance {
                material Material {
                  shininess 0.8
                }
              }
              geometry Box {
                size 0.014 0.07 0.04
              }
            }
          ]
        }
        Transform {
          translation -0.045 0 0
          rotation 0 1 0 1.5708
          children [
            USE GREY_BOX_SHAPE
          ]
        }
        Transform {
          translation 0 0 -0.0425
          children [
            USE GREY_BOX_SHAPE
          ]
        }
```

```
      Transform {
        translation 0 0 0.0435
        children [
          USE GREY_BOX_SHAPE
        ]
      }
    ]
  }
  DEF servo0 Servo {
    translation 0 0.0423 0
    rotation 0 0 1 0
    children [
      DEF GREY_BOX_TRANSFORM Transform {
        rotation 1 0 0 1.5708
        children [
          USE GREY_BOX_SHAPE
        ]
      }
    ]
    name "servo0"
    boundingObject USE GREY_BOX_TRANSFORM
    physics Physics {
      density -1
      mass 0.2
      coulombFriction 0.9
    }
    joint Joint {
      translation 0 -0.043 0
    }
    maxVelocity 6.54498
    maxForce 0.73
    maxPosition 1.5708
    minPosition -1.5708
  }
  DEF hip0 Servo {
    translation 0.089 0 0
    children [
      DEF FEMUR_GROUP Group {
        children [
          Transform {
            translation -0.031 0 0
            children [
              USE GREEN_CYLINDER_SHAPE
            ]
          }
          Transform {
            rotation 0 1 0 1.5708
            children [
              USE GREEN_BOX_SHAPE
            ]
          }
          Transform {
            translation 0.051 0 0
            rotation 1 0 0 0
            children [
              Shape {
                appearance USE ALU_APPEARANCE
```

```
                geometry Box {
                  size 0.04 0.014 0.07
                }
              }
            ]
          }
        ]
      }
      DEF knee0 Servo {
        translation 0.094 0 0
        rotation 0 0 1 0
        children [
          DEF TIBIA_GROUP Group {
            children [
              Transform {
                translation -0.031 0 0
                rotation 1 0 0 1.5708
                children [
                  USE GREEN_CYLINDER_SHAPE
                ]
              }
              Transform {
                children [
                  Shape {
                    appearance USE PLASTIC
                    geometry Box {
                      size 0.062 0.05 0.045
                    }
                  }
                ]
              }
            ]
          }
        ]
        name "knee0"
        boundingObject USE TIBIA_GROUP
        physics DEF ELEMENT_PHYSICS Physics {
          density -1
          mass 0.22
          coulombFriction 0.9
        }
        joint Joint {
          translation -0.03 0 0
        }
        maxVelocity 6.54498
        maxForce 0.73
        maxPosition 1.5708
        minPosition -1.5708
      }
    ]
    name "hip0"
    boundingObject USE FEMUR_GROUP
    physics USE ELEMENT_PHYSICS
    joint Joint {
      translation -0.031 0 0
    }
    maxVelocity 6.54498
```

```
      maxForce 0.73
      maxPosition 0.785
      minPosition -0.785
    }
    DEF hip1 Servo {
      translation 0 0 0.0855
      children [
        DEF FEMUR3_GROUP Group {
          children [
            Transform {
              translation 0 0 -0.031
              children [
                USE GREEN_CYLINDER_SHAPE
              ]
            }
            Transform {
              children [
                USE GREEN_BOX_SHAPE
              ]
            }
            Transform {
              translation 0 0 0.051
              rotation 0 0 1 1.5708
              children [
                USE GREY_BOX_SHAPE
              ]
            }
          ]
        }
        DEF knee1 Servo {
          translation 0 0 0.094
          rotation 1 0 0 0
          children [
            DEF TIBIA3_GROUP Group {
              children [
                Transform {
                  translation 0 0 -0.031
                  rotation 0 0 1 1.5708
                  children [
                    USE GREEN_CYLINDER_SHAPE
                  ]
                }
                Transform {
                  rotation 0 0 1 1.5708
                  children [
                    USE GREEN_BOX_SHAPE
                  ]
                }
              ]
            }
          ]
          name "knee1"
          boundingObject USE TIBIA3_GROUP
          physics USE ELEMENT_PHYSICS
          joint Joint {
            translation 0 0 -0.031
          }
```

```
        maxVelocity 6.54498
        maxForce 0.73
        maxPosition 1.5708
        minPosition -1.5708
      }
    ]
    name "hip1"
    boundingObject USE FEMUR3_GROUP
    physics USE ELEMENT_PHYSICS
    joint Joint {
      translation 0 0 -0.031
    }
    maxVelocity 6.54498
    maxForce 0.73
    maxPosition 0.785
    minPosition -0.785
  }
  DEF hip2 Servo {
    translation -0.089 0 0
    children [
      DEF FEMUR_GROUP Group {
        children [
          Transform {
            translation 0.031 0 0
            children [
              USE GREEN_CYLINDER_SHAPE
            ]
          }
          Transform {
            rotation 0 1 0 1.5708
            children [
              USE GREEN_BOX_SHAPE
            ]
          }
          Transform {
            translation -0.051 0 0
            rotation 1 0 0 0
            children [
              Shape {
                appearance USE ALU_APPEARANCE
                geometry Box {
                  size 0.04 0.014 0.07
                }
              }
            ]
          }
        ]
      }
      DEF knee2 Servo {
        translation -0.094 0 0
        rotation 0 0 1 0
        children [
          DEF TIBIA_GROUP Group {
            children [
              Transform {
                translation 0.031 0 0
                rotation 1 0 0 1.5708
```

```
                children [
                  USE GREEN_CYLINDER_SHAPE
                ]
              }
              Transform {
                children [
                  Shape {
                    appearance USE PLASTIC
                    geometry Box {
                      size 0.062 0.05 0.045
                    }
                  }
                ]
              }
            ]
          }
        ]
        name "knee2"
        boundingObject USE TIBIA_GROUP
        physics USE ELEMENT_PHYSICS
        joint Joint {
          translation 0.03 0 0
        }
        maxVelocity 6.54498
        maxForce 0.73
        maxPosition 1.5708
        minPosition -1.5708
      }
    ]
    name "hip2"
    boundingObject USE FEMUR_GROUP
    physics USE ELEMENT_PHYSICS
    joint Joint {
      translation 0.031 0 0
    }
    maxVelocity 6.54498
    maxForce 0.73
    maxPosition 0.785
    minPosition -0.785
  }
  DEF hip3 Servo {
    translation 0 0 -0.0855
    children [
      DEF FEMUR2_GROUP Group {
        children [
          Transform {
            translation 0 0 0.031
            children [
              USE GREEN_CYLINDER_SHAPE
            ]
          }
          Transform {
            children [
              USE GREEN_BOX_SHAPE
            ]
          }
          Transform {
```

```
            translation 0 0 -0.051
            rotation 0 0 1 1.5708
            children [
              USE GREY_BOX_SHAPE
            ]
          }
        ]
      }
      DEF knee3 Servo {
        translation 0 0 -0.094
        rotation 1 0 0 0
        children [
          DEF TIBIA2_GROUP Group {
            children [
              Transform {
                translation 0 0 0.031
                rotation 0 0 1 1.5708
                children [
                  USE GREEN_CYLINDER_SHAPE
                ]
              }
              Transform {
                rotation 0 0 1 1.5708
                children [
                  USE GREEN_BOX_SHAPE
                ]
              }
            ]
          }
        ]
        name "knee3"
        boundingObject USE TIBIA2_GROUP
        physics USE ELEMENT_PHYSICS
        joint Joint {
          translation 0 0 0.031
        }
        maxVelocity 6.54498
        maxForce 0.73
        maxPosition 1.5708
        minPosition -1.5708
      }
    ]
    name "hip3"
    boundingObject USE FEMUR2_GROUP
    physics USE ELEMENT_PHYSICS
    joint Joint {
      translation 0 0 0.031
    }
    maxVelocity 6.54498
    maxForce 0.73
    maxPosition 0.785
    minPosition -0.785
  }
]
boundingObject USE QUAD_GROUP
physics USE ELEMENT_PHYSICS
controller "walker"
```

```
}
Supervisor {
  children [
    DEF receiver Receiver {
      name "receiver"
    }
  ]
  controller "walker_reset_supervisor"
}
```

6.3 Controller configuration file

```
# configuration file for robot controller
# author: mail@yvanbourquin.com

# general
max.trials = 12000
robot.name = Tetrapod

# optimization
# 0 - parameters test
# 1 - genetic algorithm
# 2 - simulated annealing
# 3 - swarm particles
# 4 - random search
numerical.method = 3

#----- mutation parameters -----
#----- for both simulated annealing and genetic algorithm
genotype.mutation.probability = 0.06
genotype.mutation.deviation = 0.2

#----- genetic algorithm parameters -----
population.size = 100
population.elite.part = 0.05
population.sexual.part = 0.00

#----- simulated annealing parameters -----
temperature.reduction.factor = 0.95
temperature.required.successes = 5

#----- particles swarm parameters -----
initial.inertia = 1.0
inertia.reduction.factor = 0.995
swarm.size = 50
neighbourhood.size = 5
particle.speed.max = 0.2
particle.confidence.individual = 2.0
particle.confidence.social = 2.0

#----- Tetrapod -----
Tetrapod.servos.count = 8
Tetrapod.connections.count = 8
Tetrapod.start.z.rotation = 0

Tetrapod.servo[0].name = hip0
Tetrapod.servo[1].name = hip1
```

```
Tetrapod.servo[2].name = hip2
Tetrapod.servo[3].name = hip3
Tetrapod.servo[4].name = knee0
Tetrapod.servo[5].name = knee1
Tetrapod.servo[6].name = knee2
Tetrapod.servo[7].name = knee3

Tetrapod.servo[0].max = 0.785
Tetrapod.servo[1].max = 0.785
Tetrapod.servo[2].max = 0.785
Tetrapod.servo[3].max = 0.785

Tetrapod.connection[0].from = 0
Tetrapod.connection[0].to   = 1
Tetrapod.connection[1].from = 1
Tetrapod.connection[1].to   = 2
Tetrapod.connection[2].from = 2
Tetrapod.connection[2].to   = 3
Tetrapod.connection[3].from = 3
Tetrapod.connection[3].to   = 0
Tetrapod.connection[4].from = 0
Tetrapod.connection[4].to   = 4
Tetrapod.connection[5].from = 1
Tetrapod.connection[5].to   = 5
Tetrapod.connection[6].from = 2
Tetrapod.connection[6].to   = 6
Tetrapod.connection[7].from = 3
Tetrapod.connection[7].to   = 7

#----- Caterpillar -----
Caterpillar.servos.count = 6
Caterpillar.connections.count = 10
Caterpillar.start.z.rotation = 1.5708

Caterpillar.servo[0].name = servo0
Caterpillar.servo[1].name = servo1
Caterpillar.servo[2].name = servo2
Caterpillar.servo[3].name = servo3
Caterpillar.servo[4].name = servo4
Caterpillar.servo[5].name = servo5

Caterpillar.connection[0].from = 0
Caterpillar.connection[0].to   = 1
Caterpillar.connection[1].from = 1
Caterpillar.connection[1].to   = 0
Caterpillar.connection[2].from = 1
Caterpillar.connection[2].to   = 2
Caterpillar.connection[3].from = 2
Caterpillar.connection[3].to   = 1
Caterpillar.connection[4].from = 2
Caterpillar.connection[4].to   = 3
Caterpillar.connection[5].from = 3
Caterpillar.connection[5].to   = 2
Caterpillar.connection[6].from = 3
Caterpillar.connection[6].to   = 4
Caterpillar.connection[7].from = 4
Caterpillar.connection[7].to   = 3
```

```
Caterpillar.connection[8].from = 4
Caterpillar.connection[8].to   = 5
Caterpillar.connection[9].from = 5
Caterpillar.connection[9].to   = 4

#----- StiffCrawler -----
StiffCrawler.servos.count = 8
StiffCrawler.connections.count = 8
StiffCrawler.start.z.rotation = 0

StiffCrawler.servo[0].name = hip0
StiffCrawler.servo[1].name = hip1
StiffCrawler.servo[2].name = hip2
StiffCrawler.servo[3].name = hip3
StiffCrawler.servo[4].name = knee0
StiffCrawler.servo[5].name = knee1
StiffCrawler.servo[6].name = knee2
StiffCrawler.servo[7].name = knee3

StiffCrawler.servo[1].invert = 1
StiffCrawler.servo[3].invert = 1
StiffCrawler.servo[5].invert = 1
StiffCrawler.servo[7].invert = 1

StiffCrawler.connection[0].from = 0
StiffCrawler.connection[0].to   = 1
StiffCrawler.connection[1].from = 1
StiffCrawler.connection[1].to   = 3
StiffCrawler.connection[2].from = 3
StiffCrawler.connection[2].to   = 2
StiffCrawler.connection[3].from = 2
StiffCrawler.connection[3].to   = 0
StiffCrawler.connection[4].from = 0
StiffCrawler.connection[4].to   = 4
StiffCrawler.connection[5].from = 1
StiffCrawler.connection[5].to   = 5
StiffCrawler.connection[6].from = 3
StiffCrawler.connection[6].to   = 7
StiffCrawler.connection[7].from = 2
StiffCrawler.connection[7].to   = 6

#----- Wheel -----
Wheel.servos.count = 4
Wheel.connections.count = 8
Wheel.start.z.rotation = 1.5708

Wheel.servo[0].name = servo1
Wheel.servo[1].name = servo2
Wheel.servo[2].name = servo3
Wheel.servo[3].name = servo4

Wheel.servo[0].max = 0.95
Wheel.servo[1].max = 0.95
Wheel.servo[2].max = 0.95
Wheel.servo[3].max = 0.95

Wheel.connection[0].from = 0
```

```
Wheel.connection[0].to   = 1
Wheel.connection[1].from = 1
Wheel.connection[1].to   = 2
Wheel.connection[2].from = 2
Wheel.connection[2].to   = 3
Wheel.connection[3].from = 3
Wheel.connection[3].to   = 0

Wheel.connection[4].from = 1
Wheel.connection[4].to   = 0
Wheel.connection[5].from = 2
Wheel.connection[5].to   = 1
Wheel.connection[6].from = 3
Wheel.connection[6].to   = 2
Wheel.connection[7].from = 0
Wheel.connection[7].to   = 3
```

6.4 C++ listings of optimizer controller

All the listing have been written exclusively by the author with the exception of the method for generating random Gaussian numbers in the file “Random.cpp” which was taken over from Dr. Everett. F. Carter Jr., Generating Gaussian Random Numbers:
http://www.taygeta.com/random/gaussian.html
The code of the files Genotype.h, Genotype.cpp Population.h, Population.cpp was adapted from the author’s previous work for the Adaptive System course at Sussex.

6.4.1 Defaults.h

```
#ifndef Defaults_H
#define Defaults_H

/*
  Simple mechanism to configure an application using information from a file.
  Defaults works like a dictionary, every object contained is
  made of one key and its associated value.
  Author: Yvan Bourquin
*/

#include <stdio.h>

class Default;

class Defaults{
 public:
  enum Maxs {
    MAX_KEY   = 64,
    MAX_VALUE = 1024,
    MAX_LINE  = 1024
  };

  // Load all tuples from the file "fileName" into the dictionary
  // white space must separate every tokens including the = (equal) sign.
  static void loadFile(const char *fileName);

  // Retrieves the value associated with <key> in the dictionary.
  // If <key> is not found, the argument specified as default (<def>)
  // is returned.
  static const char *get(const char *key, const char *def, ...);
  static int         get(const char *key, int         def, ...);
  static float       get(const char *key, float       def, ...);

 protected:
  // constructor/destructor
  Defaults();
  ~Defaults();

 private:
  // private functions
  static Default *find(const char *, bool create = false);
  static void copy(const char *source, char *&destination);
  static bool compare(const char *a, const char *b);
  static const char *restore(const char *key);
```

```
  // linked list
  static int ourSize;
  static Default *ourHead;
  static Default *ourTail;

  Defaults(const Defaults &);
  Defaults & operator = (const Defaults &);
  // disabled
};

#endif
```

6.4.2 Defaults.cpp

```
#include "Defaults.h"
#include <stdarg.h>
#include <iostream>

using namespace std;

class Default {
public:
  // constructor
  Default(const char *nkey) {
    key     = new char[strlen(nkey) + 1];
    strcpy(key, nkey);
    user    = NULL;
    next    = NULL;
  }

public:
  char *key;
  char *user;
  Default *next;
};

int Defaults::ourSize = 0;
Default *Defaults::ourHead = NULL;
Default *Defaults::ourTail = NULL;

// string copy with memory managment and NULL pointers allowed
void Defaults::copy(const char *source, char *&destination) {
  delete [] destination;

  if (! source) {
    destination = NULL;
    return;
  }

  destination = new char[strlen(source) + 1];
  strcpy(destination, source);
}

// string compare with NULL pointers allowed.
bool Defaults::compare(const char *a, const char *b) {
  if (! a && ! b)
```

```
    return true;

  if (! a || ! b)
    return false;

  return strcmp(a, b) ? false : true;
}

// load dictionnary from file
void Defaults::loadFile(const char *filename) {
  FILE *file = fopen(filename, "r");
  if (file == NULL) {
    cerr << "Defaults::loadFile(): failed to open file.\n";
    return;
  }

  while (! feof(file)) {
    char line[MAX_LINE], *linep;

    // read one line
    fgets(line, sizeof(line), file);
    linep = line;

    // scan line forward skipping comments
    while (*linep != '#' && *linep != '\n' && *linep != '\0')
      linep++;

    // scan line backward skipping trailing spaces
    while (linep > line && isspace(*(linep - 1)))
      linep--;

    // mark end of line
    *linep = '\0';

    // if line is not empty...
    if (line[0] != '\0') {
      linep = line;

      // skip extra spaces
      while (isspace(*linep))
     linep++;

      // extract <key>
      char *key = linep;
      while (*linep != '=' && ! isspace(*linep) && *linep != '\0')
     linep++;

      *linep++ = '\0';

      // skip '=' and extra spaces and extract <value>
      while (*linep == '=' || isspace(*linep))
     linep++;

      char *value = linep;

      // store value
      Default *defp = find(key, true);
```

```
      copy(value, defp->user);
    }
  }

  fclose(file);
}

// find tuple in dictionary
Default *Defaults::find(const char *key, bool create) {
  for (Default *defp = ourHead; defp; defp = defp->next)
    if (compare(key, defp->key))
      return defp;

  if (create) {
    Default *newp = new Default(key);
    // append at the tail
    if (ourTail)
      ourTail->next = newp;
    ourTail = newp;
    if (! ourHead)
      ourHead = newp;
    ourSize++;
    return newp;
  }

  return NULL;
}

const char *Defaults::get(const char *format, const char *def, ...) {
  va_list ap;
  va_start(ap, def);

  if (! format)
    return def;

  char key[MAX_KEY];
  vsprintf(key, format, ap);

  const char *value = restore(key);
  if (value) {
    if (! strcmp(value, "NULL"))
      def = NULL;
    else
      def = value;
  }

  va_end(ap);
  return def;
}

int Defaults::get(const char *format, int def, ...) {
  va_list ap;
  va_start(ap, def);

  if (! format)
    return def;
```

```
  char key[MAX_KEY];
  vsprintf(key, format, ap);

  const char *value = restore(key);
  if (value)
    def = atoi(value);

  va_end(ap);
  return def;
}

float Defaults::get(const char *format, float def, ...) {
  va_list ap;
  va_start(ap, def);

  if (! format)
    return def;

  char key[MAX_KEY];
  vsprintf(key, format, ap);

  const char *value = restore(key);
  if (value)
    def = atof(value);

  va_end(ap);
  return def;
}

const char *Defaults::restore(const char *key) {
  Default *defp = find(key);
  if (! defp)
    return NULL;

  return defp->user;
}
```

6.4.3 GeneticAlgorithm.h

```
#ifndef GeneticAlgorithm_H
#define GeneticAlgorithm_H

/*
  Genetic algorithm for robotic locomotion optimization
  Author: Yvan Bourquin
*/

class Population;
class Robot;

#include "Optimizer.h"

class GeneticAlgorithm : public Optimizer {
 public:
  // constructor
  GeneticAlgorithm(Robot *robot);
```

```
  // destructor
  virtual ~GeneticAlgorithm();

  // run genetic algorithm
  virtual void run();

 private:
  Population *_population;    // genotype population
  char _filename[64];         // data file
  int _popSize;               // population size

  void save(const char *filename) const;
};

#endif
```

6.4.4 GeneticAlgorithm.cpp

```
#include "GeneticAlgorithm.h"
#include "Robot.h"
#include "Population.h"
#include "Defaults.h"
#include <iostream>

using namespace std;

GeneticAlgorithm::GeneticAlgorithm(Robot *robot)
  : Optimizer(robot) {
  _popSize = Defaults::get("population.size", 100);

  sprintf(_filename, "../../results/ga_%d_%s%d.m", _popSize,
       robot->getName(), getNumParams());

  cout << "filename: " << _filename << endl;
}

GeneticAlgorithm::~GeneticAlgorithm() {
  delete _population;
}

void GeneticAlgorithm::save(const char *filename) const {
  FILE *file = fopen(filename, "w");
  if (! file) {
    printf("could not write file: %s\n", filename);
    return;
  }

  cout << "writing file: " << filename << endl;

  Optimizer::save(file);
  _population->save(file);

  fclose(file);
}

void GeneticAlgorithm::run() {
  _population = new Population(_popSize, getNumParams());
```

```
  cout << "starting genetic algorithm ...\n"
       << "population size is "
       << _popSize << ", genome size is "
       << getNumParams() << endl;

  while (! isFinished()) {
    for (int i = 0; i < _population->getSize(); i++) {

      cout << "evaluating generation: " << _population->getGeneration()
        << " genotype: " << i << endl;

      // evaluate genotype
      const float *genome = _population->getGenome(i);
      float fitness = Optimizer::evaluate(genome);
      _population->setFitness(i, fitness);
    }

    _population->sort();
    Genotype *fittest = _population->getFittest();
    Optimizer::recordSolution(fittest->getFitness(), fittest->getGenes());
    save(_filename);
    _population->reproduce();
  }
}
```

6.4.5 Genotype.h

```
#ifndef Genotype_H
#define Genotype_H

/*
  General-purpose genotype with mutation and crossover operations
  Author: Yvan Bourquin
*/

#include <stdio.h>
#include <iostream>

using namespace std;

class Genotype {
 public:

  // constructor
  Genotype(int size);

  // copy constructor
  Genotype(const Genotype &);

  // destructor
  virtual ~Genotype();

  // assignment operator
  Genotype &operator = (const Genotype &);

  // mutation
```

```
  void hypersphereMutate();
  void singleLocusMutate();

  // crossover
  Genotype crossover(const Genotype &other) const;

  // set/get fitness
  void setFitness(float fitness) { _fitness = fitness; }
  float getFitness() const { return _fitness; }

  // get array of floating points
  const float *getGenes() const { return _genes; }

  // write genotype to file
  void save(FILE *file) const;

  // class wide mutation parameters
  static float getMutationProbability();
  static float getMutationDeviation();

 private:
  float *_genes;    // genome
  int _size;        // genome length
  float _fitness;

  static float _mutationProbability;
  static float _mutationDeviation;
};

#endif
```

6.4.6 Genotype.cpp

```
#include "Genotype.h"
#include "Random.h"
#include "Defaults.h"
#include <math.h>

float Genotype::_mutationProbability = -1.0;
float Genotype::_mutationDeviation = -1.0;

float Genotype::getMutationProbability() {
  if (_mutationProbability < 0.0)
    _mutationProbability = Defaults::get("genotype.mutation.probability", 0.1f);

  return _mutationProbability;
}

float Genotype::getMutationDeviation() {
  if (_mutationDeviation < 0.0)
    _mutationDeviation = Defaults::get("genotype.mutation.deviation", 0.4f);

  return _mutationDeviation;
}

Genotype::Genotype(int size)
  : _size(size), _fitness(0.0) {
```

```
  _genes = new float[_size];

  // initialize with random uniform numbers in the range [0,1]
  for (int i = 0; i < _size; i++)
    _genes[i] = Random::getUniform();
}

Genotype::Genotype(const Genotype &other)
  : _size(other._size), _fitness(other._fitness) {
  _genes = new float[_size];

  for (int i = 0; i < _size; i++)
    _genes[i] = other._genes[i];
}

Genotype &Genotype::operator = (const Genotype &other) {
  // avoid crash in case of inadvertant: a = a
  if (&other == this)
    return *this;

  delete [] _genes;

  _size = other._size;
  _genes = new float[_size];
  _fitness = other._fitness;

  for (int i = 0; i < _size; i++)
    _genes[i] = other._genes[i];

  return *this;
}

Genotype::~Genotype() {
  delete [] _genes;
}

void Genotype::hypersphereMutate() {
  float length = Random::getGaussian() * getMutationDeviation();
  float *mutation = new float[_size];

  float sum = 0.0;
  for (int i = 0; i < _size; i++) {
    mutation[i] = Random::getUniform();
    sum += mutation[i] * mutation[i];
  }

  float ratio = length / (float)sqrt(sum);
  for (int i = 0; i < _size; i++) {
    _genes[i] += mutation[i] * ratio;

    // check range
    if (_genes[i] > 1.0)
      _genes[i] = 1.0;
    else if (_genes[i] < 0.0)
      _genes[i] = 0.0;
  }
```

```
  delete [] mutation;
}

// mutate a every gene with given probability and deviation
void Genotype::singleLocusMutate() {
  for (int i = 0; i < _size; i++)
    if (Random::getUniform() < getMutationProbability()) {
      _genes[i] += Random::getGaussian() * getMutationDeviation();

      // check range
      if (_genes[i] > 1.0)
     _genes[i] = 1.0;
      else if (_genes[i] < 0.0)
     _genes[i] = 0.0;
    }
}

// single-point crossover
Genotype Genotype::crossover(const Genotype &other) const {
  Genotype child(_size);

  // make sure we don't always start with the same parent
  const float *mom, *dad;
  if (Random::getInteger(2) == 0) {
    mom = this->_genes;
    dad = other._genes;
  }
  else {
    mom = other._genes;
    dad = this->_genes;
  }

  int locus = Random::getInteger(_size);

  for (int i = 0; i < _size; i++)
    if (i < locus)
      child._genes[i] = mom[i];
    else
      child._genes[i] = dad[i];

  return child;
}

void Genotype::save(FILE *file) const {
  fprintf(file, "%.3f ", _fitness);

  for (int i = 0; i < _size; i++)
    fprintf(file, "%.4f ", _genes[i]);

  fprintf(file, ";\n");
}
```

6.4.7 Main.cpp

```
#include "Robot.h"
#include "GeneticAlgorithm.h"
#include "SimulatedAnnealing.h"
```

```
#include "ParticleSwarm.h"
#include "RandomSearch.h"
#include "Defaults.h"
#include <device/robot.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>

static Robot *robot = NULL;

using namespace std;

void parametersTest(Robot *robot) {
  const char *paramString = Defaults::get("parameters", (const char*)NULL);
  if (! paramString) {
    cout << "missing test parameters ! exiting ...\n";
    return;
  }

  float *params = new float[robot->getNumParams()];
  int p = 0;
  for (int i = 0; i < robot->getNumParams(); i++) {
    int n;
    sscanf(paramString + p, "%f%n", &params[i], &n);
    p += n;
  }

  cout << "starting parameters test ...\nparameters: ";

  for (int i = 0; i < robot->getNumParams(); i++)
    cout << params[i] << " ";

  cout << endl;

  while (true)
    robot->evaluate(params);

  delete params;
}

void reset(void) {
  const char *robotName = Defaults::get("robot.name", "robot");
  robot = new Robot(robotName);
}

int main(int argc, char *argv[]) {
  srand(time(NULL));
  Defaults::loadFile("../walker.ini");

  robot_live(reset);

  int method = Defaults::get("numerical.method", -1);
  if (method == 0) {
    parametersTest(robot);
  }
  else if (method == 1) {
    GeneticAlgorithm *ga = new GeneticAlgorithm(robot);
```

```
    ga->run();
    delete ga;
  }
  else if (method == 2) {
    SimulatedAnnealing *sa = new SimulatedAnnealing(robot);
    sa->run();
    delete sa;
  }
  else if (method == 3) {
    ParticleSwarm *ps = new ParticleSwarm(robot);
    ps->run();
    delete ps;
  }
  else if (method == 4) {
    RandomSearch *rs = new RandomSearch(robot);
    rs->run();
    delete rs;
  }
  else
    cerr << "no numerical search method specified\n";

  delete robot;
  return 0;
}
```

6.4.8 Module.h

```
#ifndef Module_H
#define Module_H

/*
  Module of modular robot
  Author: Yvan Bourquin
*/

class Oscillator;

#include <device/robot.h>

class Module {
 public:
  // constructor: construct module with "numInputs" oscillator inputs
  // for robot servo "devtag"
  Module(const char *robotName,int servoNum, int numInputs);

  // destructor
  virtual ~Module();

  // add oscillator input from specified module
  void addInput(const Module *module);

  // reset oscillator and servo
  const float *reset(const float *params);

  // evaluate oscillator and update servo position
  void step();
```

```
  // return true if the oscialltor is chaotic
  bool isCrazy() const;

 private:
  DeviceTag _servo;        // module's servo motor
  Oscillator *_oscillator; // module's oscillator
  float _x0;               // initial servo angle
  float _maxx;             // maximal servo angle allowed
  float _invert;           // invert servo rotation angle
};

#endif
```

6.4.9 Module.cpp

```
#include "Module.h"
#include "Oscillator.h"
#include "Defaults.h"
#include <device/servo.h>
#include <cmath>
#include <iostream>

using namespace std;

static const float DEFAULT_MAX_ANGLE = M_PI / 2.0; // 90 degrees

Module::Module(const char *robotName, int servoNum, int numInputs) {

  const char *servoName =
    Defaults::get("%s.servo[%d].name",
            (const char*)NULL, robotName, servoNum);

  // get device tag from simulator
  _servo = robot_get_device(servoName);
  if (! _servo) {
    cout << "ERROR: could not find servo: " << servoName << endl;
    exit(0);
  }

  cout << "found servo: " << servoName << endl;

  // max servo angle according to physical robot properties
  _maxx = Defaults::get("%s.servo[%d].max",
              DEFAULT_MAX_ANGLE, robotName, servoNum);

  _invert = Defaults::get("%s.servo[%d].invert",
                0, robotName, servoNum) == 0 ? 1.0 : -1.0;

  servo_set_velocity(_servo, 1.0);
  servo_set_force(_servo, 100.0);

  // create oscillator with max possible oscillation amplitude
  _oscillator = new Oscillator(numInputs);
  _x0 = 0.0;
}

Module::~Module() {
```

```
  delete _oscillator;
}

const float *Module::reset(const float *params) {
  _x0 = *params++ * (2.0 * _maxx) - _maxx;
  return _oscillator->reset(params);
}

void Module::step() {
  _oscillator->step();

  if (_oscillator->isDiverging())
    return;

  float x = (_oscillator->getX() + _x0) * _invert;

  // constrict output to allowed range because we
  // don't want to force the servo motors
  if (x > _maxx)
    x = _maxx;
  else if (x < -_maxx)
    x = -_maxx;

  // now move servo to desired position
  servo_set_position(_servo, x);
}

void Module::addInput(const Module *module) {
  _oscillator->addInput(module->_oscillator);
}

bool Module::isCrazy() const {
  return _oscillator->isDiverging();
}
```

6.4.10 Optimizer.h

```
#ifndef Optimizer_H
#define Optimizer_H

/*
  Base class for numerical optimization methods
  Author: Yvan Bourquin
*/

#include <stdio.h>

class Robot;

class Optimizer {
 public:

  // number of simulation trials in an optimization
  int getNumTrials() const { return _nTrials; }

  // number of parameters of the robot controller
  int getNumParams() const { return _nParams; }
```

```
  // the robot
  Robot *getRobot() const { return _robot; }

  // start the optimization (pure virtual method)
  virtual void run() = 0;

 protected:
  // constructor: create optimization procedure
  Optimizer(Robot *robot);

  // destructor
  virtual ~Optimizer();

  // return true when the max number of trials was reached
  bool isFinished() const { return _finished; }

  // evaluate robot controller parameters
  // and return the corresponding fitness
  float evaluate(const float *params);

  // record the parameters as solution of the optimization process
  void recordSolution(float fitness, const float *params);

  // save results to a file
  void save(FILE *) const;

 private:
  int _maxTrials;     // max muber of evaluation trials
  Robot *_robot;      // the robot
  int _trials[1000];  // trial number of every fitness change
  float _fitb[1000];  // fitness change versus trial number
  int _nData;         // number of fitness improvement data
  int _nTrials;       // number of trials so far
  int _nParams;       // number of parameters for the robot controller
  float *_best;       // best solution so far
  bool _finished;     // optimization finished flag
  int _nSolutions;    // number of recorded solutions
  float *_solution;   // best solution
  float _fits[1000];  // fitness of solutions
  int _strials[1000]; // trial number of solutions

  void recordTrial(float fitness, const float *params);
};

#endif
```

6.4.11 Optimizer.cpp

```
#include "Optimizer.h"
#include "Defaults.h"
#include "Robot.h"

Optimizer::Optimizer(Robot *robot) {
  _maxTrials = Defaults::get("max.trials", 10000);
  _robot = robot;
  _nParams = robot->getNumParams();
```

```
  _best = new float[_nParams];
  _solution = new float[_nParams];
  _nData = 0;
  _nTrials = 0;
  _finished = false;
  _nSolutions = 0;
}

Optimizer::~Optimizer() {
  delete [] _best;
  delete [] _solution;
}

float Optimizer::evaluate(const float *params) {
  float fitness = _robot->evaluate(params);
  recordTrial(fitness, params);
  return fitness;
}

void Optimizer::recordTrial(float fitness, const float *params) {

  _nTrials++;

  if (_nData == 0 || fitness > _fitb[_nData - 1]) {
    _trials[_nData] = _nTrials;
    _fitb[_nData] = fitness;
    _nData++;
    for (int i = 0; i < _nParams; i++)
      _best[i] = params[i];
  }

  if (_nTrials >= _maxTrials)
    _finished = true;
}

void Optimizer::recordSolution(float fitness, const float *params) {
  _fits[_nSolutions] = fitness;
  _strials[_nSolutions] = _nTrials;
  _nSolutions++;

  for (int i = 0; i < _nParams; i++)
    _solution[i] = params[i];
}

void Optimizer::save(FILE *file) const {
  fprintf(file, "nparams= %d ;\nntrials= %d ;\nnimp= %d ;\nfitimp=[ ",
       _nParams, _nTrials, _nData);

  for (int i = 0; i < _nData; i++)
    fprintf(file, "%.3f ", _fitb[i]);

  fprintf(file, "];\ntrialimp=[ ");

  for (int i = 0; i < _nData; i++)
    fprintf(file, "%d ", _trials[i]);

  fprintf(file, "];\nbest=[ ");
```

```
  for (int i = 0; i < _nParams; i++)
    fprintf(file, "%.6f ", _best[i]);

  fprintf(file, "];\nnsolutions= %d ;\nfitsol=[ ", _nSolutions);

  for (int i = 0; i < _nSolutions; i++)
    fprintf(file, "%.3f ", _fits[i]);

  fprintf(file, "];\ntrialsol= [ ");

  for (int i = 0; i < _nSolutions; i++)
    fprintf(file, "%d ", _strials[i]);

  fprintf(file, "];\nsolution=[ ");

  for (int i = 0; i < _nParams; i++)
    fprintf(file, "%.6f ", _solution[i]);

  fprintf(file, "];\n");
}
```

6.4.12 Oscillator.h

```
#ifndef Oscillator_H
#define Oscillator_H

/*
  Non-linear oscillator for robot controller
  Author: Yvan Bourquin
*/

class Oscillator {
 public:
  // constructor
  //Oscillator(int numInputs, float maxAmplitude);
  Oscillator(int numInputs);

  // destructor
  virtual ~Oscillator();

  // connect output from other oscillators
  void addInput(const Oscillator *connection);

  // reset to initial parameters
  const float *reset(const float *params);

  // compute one step
  void step();

  // current oscillator outputs
  float getX() const { return _x; }
  float getV() const { return _v; }

  // is showing chaotic behaviour
  bool isDiverging() const;
```

```
 private:
  float _x;              // current x
  float _v;              // current v
  float _E;              // energy
  int _numInputs;        // number of inputs
  Oscillator **_inputs;  // inputs
  float *_a;             // connection strength a
  float *_b;             // connection strength b
};

#endif
```

6.4.13 Oscillator.cpp

```
#include "Oscillator.h"
#include "Random.h"
#include <cmath>
#include <stdio.h>

using namespace std;

static const float F = 1.0;                // 1Hz
static const float ALPHA = 0.5;            // convergence speed
static const float TAU = F / (2.0 * M_PI);  // tends to -> 1Hz
static const float DT = 0.0001f;           // integration step [s]
static const float ROBOT_SIMULATION_STEP = 0.064;   // [s]
static const float SEQUENCE_SIZE = ROBOT_SIMULATION_STEP / DT;
static const float AMPLITUDE_MIN = 0.0001f;
static const float AMPLITUDE_MAX = M_PI / 2.0;
static const float CONNECTION_MIN = -0.7;  // min connection strength
static const float CONNECTION_MAX = +0.7;  // max connection strength

Oscillator::Oscillator(int numInputs) {
  //_maxAmplitude = maxAmplitude;
  _numInputs = 0;
  _v = 0.1;
  _x = 0.1;
  _E = 1.0;
  _inputs = new Oscillator *[numInputs];
  _a = new float[numInputs];
  _b = new float[numInputs];
}

Oscillator::~Oscillator() {
  delete [] _b;
  delete [] _a;
  delete [] _inputs;
}

void Oscillator::step() {
  for (int i = 0; i < SEQUENCE_SIZE; i++) {
    // compute current sum of external inputs
    float sum = 0.0;
    for (int i = 0; i < _numInputs; i++) {
      float x = _inputs[i]->getX();
      float v = _inputs[i]->getV();
```

```
      // add normalized input
      sum += (_a[i] * x + _b[i] * v) / (x * x + v * v);
    }

    // compute new oscillator state according to difference equation
    _v = _v + DT * ((-ALPHA * ((_x * _x + _v * _v - _E) / _E) * _v - _x + sum) /
TAU);
    _x = _x + DT * _v / TAU;
  }
}

// setup oscillator according to given parameters
// this is done once, at the beginning of every evaluation
const float *Oscillator::reset(const float *params) {

  // reset oscillator state
  _v = 0.1;
  _x = 0.1;

  // decode amplitude
  float A = *params++ * (AMPLITUDE_MAX - AMPLITUDE_MIN) + AMPLITUDE_MIN;
  _E = A * A;

  // decode a[] connections
  for (int i = 0; i < _numInputs; i++)
    _a[i] = *params++ * (CONNECTION_MAX - CONNECTION_MIN) + CONNECTION_MIN;

  // decode b[] connections
  for (int i = 0; i < _numInputs; i++)
    _b[i] = *params++ * (CONNECTION_MAX - CONNECTION_MIN) + CONNECTION_MIN;

  return params;
}

// add oscillator input
// this is done only once per simulation
void Oscillator::addInput(const Oscillator *input) {
  _inputs[_numInputs++] = const_cast<Oscillator*>(input);
}

bool Oscillator::isDiverging() const {
  return fabsf(_x) > 50.0f || fabsf(_v) > 50.0f || isnan(_x) || isnan(_v);
}
```

6.4.14 Particle.h

```
#ifndef Particle_H
#define Particle_H

/*
  Particle for Particle Swarm Optimization (PSO)
  Author: Yvan Bourquin
 */

#include <stdio.h>

class Particle {
```

```
 public:
  // constructor: create particle
  Particle(int numParams);

  // destructor
  virtual ~Particle();

  // distance between 2 particles
  float distanceTo(const Particle *particle) const;

  // compute next position according to PSO parameters:
  // best overall position and current inertia
  void computeNextPosition(const float *bestNeighbourParams, float inertia);

  // the previously computed position becomes the current position
  void updatePosition();

  // set particle fitness after measurement
  void setFitness(float fitness);
  float getFitness() const { return _fitness; }

  // return best fitness encountered so far by this particle
  float getBestFitness() const { return _bestFitness; }

  // return current position
  const float *getParams() const { return _params; }

  // return best position encountered so far by this particle
  const float *getBestParams() const { return _bestParams; }

  // write particle to file
  void save(FILE *file) const;

  // get class parameters
  static float getMaxSpeed() { return _maxSpeed; }
  static float getB1() { return _b1; }
  static float getB2() { return _b2; }

 private:
  int _numParams;      // number of parameters in the robot controller
  float *_params;      // particle's current position
  float _fitness;      // fitness of current position
  float *_nextParams;  // position for next iteration
  float *_speed;       // current speed in parameter space
  float *_bestParams;  // best position encountered so far
  float _bestFitness;  // fitness of best position

  static float _maxSpeed;
  static float _b1;
  static float _b2;
};

#endif
```

6.4.15 Particle.cpp

```
#include "Particle.h"
```

```
#include "Random.h"
#include "Defaults.h"
#include <cmath>

float Particle::_maxSpeed = -1.0;  // speed limit
float Particle::_b1 = -1.0;        // individual confidence factor
float Particle::_b2 = -1.0;        // social confidence factor

Particle::Particle(int numParams) {

  // if not assigned yet ...
  if (_maxSpeed < 0.0) {
    // get data from configuration file
    _maxSpeed = Defaults::get("particle.speed.max", 0.2f);
    _b1 = Defaults::get("particle.confidence.individual", 2.0f);
    _b2 = Defaults::get("particle.confidence.social", 2.0f);
  }

  _numParams = numParams;
  _params = new float[numParams];     // current position
  _nextParams = new float[numParams]; // position of next time step
  _bestParams = new float[numParams]; // best position so far
  _speed = new float[numParams];

  // start initializing positions and speed at random
  for (int i = 0; i < numParams; i++) {
    _params[i] = Random::getUniform();
    _bestParams[i] = _params[i];
    _speed[i] = Random::getUniform() * 2.0 * _maxSpeed - _maxSpeed;
  }

  _fitness = 0.0;
  _bestFitness = 0.0;
}

Particle::~Particle() {
  delete [] _params;
  delete [] _nextParams;
  delete [] _bestParams;
  delete [] _speed;
}

void Particle::save(FILE *file) const {
  fprintf(file, "%.3f %.3f ", _fitness, _bestFitness);

  for (int i = 0; i < _numParams; i++)
    fprintf(file, "%.4f ", _params[i]);

  for (int i = 0; i < _numParams; i++)
    fprintf(file, "%.4f ", _speed[i]);
}

void Particle::setFitness(float fitness) {
  _fitness = fitness;

  if (fitness > _bestFitness) {
    _bestFitness = fitness;
```

```
    for (int i = 0; i < _numParams; i++)
      _bestParams[i] = _params[i];
  }
}

// compute distance between 2 particles in the parameter space
float Particle::distanceTo(const Particle *particle) const {
  float distance = 0.0;
  for (int i = 0; i < _numParams; i++) {
    float diff = this->_params[i] - particle->_params[i];
    distance += diff * diff;
  }

  return sqrt(distance);
}

// compute next particle speed and position according to basic PSO rules
void Particle::computeNextPosition(const float *bestNeighbourParams,
                      float inertia) {

  for (int i = 0; i < _numParams; i++) {

    // compute new speed
    float c1 = Random::getUniform() * _b1;
    float c2 = Random::getUniform() * _b2;
    _speed[i] = inertia * _speed[i] + c1 * (_bestParams[i] - _params[i])
      + c2 * (bestNeighbourParams[i] - _params[i]);

    // constrict speed
    if (_speed[i] > _maxSpeed)
      _speed[i] = _maxSpeed;
    else if (_speed[i] < -_maxSpeed)
      _speed[i] = -_maxSpeed;
  }

  // next position
  for (int i = 0; i < _numParams; i++) {

    // constrict position
    _nextParams[i] = _params[i] + _speed[i];
    if (_nextParams[i] < 0.0) {
      _nextParams[i] = 0.0;
      _speed[i] = -_speed[i];
    }
    else if (_nextParams[i] > 1.0) {
      _nextParams[i] = 1.0;
      _speed[i] = -_speed[i];
    }
  }
}

// current position = next position
void Particle::updatePosition() {
  for (int i = 0; i < _numParams; i++)
    _params[i] = _nextParams[i];
}
```

6.4.16 ParticleSwarm.h

```
#ifndef ParticleSwarm_H
#define ParticleSwarm_H

/*
  Particle Swarm Optimization for robot locomotion
  Author: Yvan Bourquin
*/

#include "Optimizer.h"

class Robot;
class Particle;

class ParticleSwarm : public Optimizer {
 public:

  // constructior: setup particle swarm optimization procedure
  ParticleSwarm(Robot *robot);

  // desctructor
  virtual ~ParticleSwarm();

  // run particle swarm optimization
  virtual void run();

 private:
  int _neighbourhoodSize;  // particle neighbourhood size
  float _reductionFactor;  // inertia reduction factor
  float _maxSpeed;         // max particle speed in parameter n-space
  int _iteration;          // current algorithm iteration
  char _filename[64];      // file name
  int _size;               // num particles in swarm
  Particle **_particles;   // particles
  float **_distances;      // inter-particle distance matrix
  float _inertia;          // current inertia
  float *_bests;           // record of best fitness
  float *_means;           // record of mean fitness

  void save(const char *fileName) const;
  Particle *findBestNeighbourOf(int particleIndex) const;
  Particle *findBestOverall() const;
  void evaluateSwarm();
  void computeDistances();
};

#endif
```

6.4.17 ParticleSwarm.cpp

```
#include "ParticleSwarm.h"
#include "Particle.h"
#include "Robot.h"
#include "Defaults.h"
#include <limits>
#include <iostream>
```

```
using namespace std;

ParticleSwarm::ParticleSwarm(Robot *robot)
  : Optimizer(robot) {

  _size = Defaults::get("swarm.size", 50);
  _particles = new Particle*[_size];
  _bests = new float[2000];
  _means = new float[2000];
  _inertia = Defaults::get("initial.inertia", 1.0f);

  for (int i = 0; i < _size; i++)
    _particles[i] = new Particle(getNumParams());

  _distances = new float*[_size];
  for (int i = 0; i < _size; i++)
    _distances[i] = new float[_size];

  _neighbourhoodSize = Defaults::get("neighbourhood.size", 5);
  _reductionFactor = Defaults::get("inertia.reduction.factor", 0.995f);

  sprintf(_filename, "../../results/pso_%d_%d_%s%d.m", _size,
       _neighbourhoodSize, robot->getName(), getNumParams());

  cout << "filename: " << _filename << endl;
}

ParticleSwarm::~ParticleSwarm() {
  for (int i = 0; i < _size; i++)
    delete _particles[i];

  delete [] _particles;

  for (int i = 0; i < _size; i++)
    delete [] _distances[i];

  delete [] _distances;
  delete [] _bests;
  delete [] _means;
}

void ParticleSwarm::save(const char *filename) const {
  FILE *file = fopen(filename, "w");
  if (! file) {
    printf("could not write file: %s\n", filename);
    return;
  }

  cout << "writing file: " << filename << endl;

  Optimizer::save(file);

  fprintf(file, "inertia= %.5f ;\nnparticles= %d ;\n"
       "maxspeed= %.2f ;\nb1= %.2f ;\nb2= %.2f ;\nbests=[ ",
       _inertia, _size, Particle::getMaxSpeed(),
       Particle::getB1(), Particle::getB2());
```

```
  for (int i = 0; i < _iteration + 1; i++)
    fprintf(file, "%.3f ", _bests[i]);

  fprintf(file, "];\nmeans=[ ");

  for (int i = 0; i < _iteration + 1; i++)
    fprintf(file, "%.3f ", _means[i]);

  fprintf(file, "];\nparticles=[\n");

  for (int i = 0; i < _size; i++) {
    _particles[i]->save(file);
    fprintf(file, ";\n");
  }

  fprintf(file, "];\n");
  fclose(file);
}

void ParticleSwarm::run() {

  cout << "starting swarm particle optimization ...\n";

  // PSO main loop
  for (_iteration = 0; ! isFinished(); _iteration++) {

    // evaluate fitness of every particle
    evaluateSwarm();

    // compute new inter-particle distances
    computeDistances();

    // PSO update step for all particles
    for (int i = 0; i < _size; i++) {

      // find neighbour with best fitness
      Particle *bestNeighbour = findBestNeighbourOf(i);

      // compute next position
      _particles[i]->computeNextPosition(bestNeighbour->getBestParams(),
_inertia);
    }

    Particle *best = findBestOverall();
    recordSolution(best->getBestFitness(), best->getBestParams());
    save(_filename);

    // apply next position synchronously
    for (int i = 0; i < _size; i++)
      _particles[i]->updatePosition();

    _inertia *= _reductionFactor;
  }
}

Particle *ParticleSwarm::findBestOverall() const {
```

```
  Particle *bestParticle = _particles[0];

  for (int i = 1; i < _size; i++)
    if (_particles[i]->getBestFitness() > bestParticle->getBestFitness())
      bestParticle = _particles[i];

  return bestParticle;
}

Particle *ParticleSwarm::findBestNeighbourOf(int particleIndex) const {
  Particle *neighbours[_neighbourhoodSize];

  cout << "minDistances: ";

  // find closest neighbours ...
  for (int j = 0; j < _neighbourhoodSize; j++) {

    // find closest neighbour ...
    float min = numeric_limits<float>::max();
    int minIndex = 0;
    for (int k = 0; k < _size; k++) {
      if (_distances[particleIndex][k] < min) {
     min = _distances[particleIndex][k];
     minIndex = k;
      }
    }

    cout << min << " ";

    neighbours[j] = _particles[minIndex];
    _distances[particleIndex][minIndex] = numeric_limits<float>::max();
  }

  cout << endl;

  // find neighbour with best fitness
  Particle *bestNeighbour = neighbours[0];
  for (int j = 1; j < _neighbourhoodSize; j++)
    if (_particles[j]->getFitness() > bestNeighbour->getBestFitness())
      bestNeighbour = _particles[j];

  return bestNeighbour;
}

void ParticleSwarm::evaluateSwarm() {

  _bests[_iteration] = numeric_limits<float>::min();
  _means[_iteration] = 0.0;
  for (int i = 0; i < _size; i++) {

    cout << "iteration: " << _iteration
      << " inertia: " << _inertia
      << " particle: " << i << endl;

    float fitness = Optimizer::evaluate(_particles[i]->getParams());
    _particles[i]->setFitness(fitness);
```

```
    if (fitness > _bests[_iteration])
      _bests[_iteration] = fitness;

    _means[_iteration] += fitness;
  }

  _means[_iteration] /= _size;
}

void ParticleSwarm::computeDistances() {
  // compute inter-particle distances
  for (int i = 0; i < _size; i++) {
    for (int j = 0; j < i; j++) {
      float distance = _particles[i]->distanceTo(_particles[j]);
      _distances[i][j] = distance;
      _distances[j][i] = distance;
    }

    _distances[i][i] = 0.0;
  }
}
```

6.4.18 Population.h

```
#ifndef Population_H
#define Population_H

/*
  Genotype population with selection and crossover for genetic algorithm
  Author: Yvan Bourquin
*/

#include "Genotype.h"

class Population
{
 public:
  // constructor: create population with specified size and genome size
  Population(int popSize, int genSize);

  // destructor
  virtual ~Population();

  // population size
  int getSize() const { return _size; }

  // save to file
  void save(FILE *file) const;

  // sort population from most fit to least fit individual
  void sort();

  // fittest
  Genotype *getFittest() const
    { return _genotypes[0]; }

  // change generation
```

```
  // precondition: population must be sorted before this function is called
  void reproduce();

  // current generation number
  int getGeneration() const { return _generation; }

  // get genetic code
  const float *getGenome(int index) const
    { return _genotypes[index]->getGenes(); }

  // get/set fitness
  void setFitness(int index, float fitness)
    { _genotypes[index]->setFitness(fitness); }

  float getFitness(int index) const
    { return _genotypes[index]->getFitness(); }

 private:
  float _elitePart;      // propotion of population cloned (no mutation)
  float _sexualPart;     // proportion of population reproducing sexually
  Genotype **_genotypes; // genotypes
  int _size;             // population size
  int _genotypeSize;     // size of each genotype
  int _generation;       // current generation
  float *_meanFitness;   // record of mean fitness
  float *_bestFitness;   // record of best fitness

  const Genotype *selectParent() const;
};

#endif
```

6.4.19 Population.cpp

```
#include "Population.h"
#include "Random.h"
#include "Defaults.h"
#include <stdlib.h>
#include <cassert>
#include <iostream>

Population::Population(int populationSize, int genotypeSize) {

  _elitePart = Defaults::get("population.elite.part", 0.05f);
  _sexualPart = Defaults::get("population.sexual.part", 0.05f);
  _size = populationSize;
  _genotypeSize = genotypeSize;
  _generation = 0;
  _genotypes = new Genotype*[_size];
  _meanFitness = new float[1000];
  _bestFitness = new float[1000];

  for (int i = 0; i < _size; i++)
    _genotypes[i] = new Genotype(genotypeSize);
}

Population::~Population() {
```

```
  delete [] _bestFitness;
  delete [] _meanFitness;

  for (int i = 0; i < _size; i++)
    delete _genotypes[i];

  delete [] _genotypes;
}

void Population::save(FILE *file) const {

  fprintf(file, "generation= %d ;\nbests=[ ", _generation + 1);

  for (int i = 0; i < _generation + 1; i++)
    fprintf(file, "%.3f ", _bestFitness[i]);

  fprintf(file, "];\nmeans=[ ");

  for (int i = 0; i < _generation + 1; i++)
    fprintf(file, "%.3f ", _meanFitness[i]);

  fprintf(file, "];\npopsize= %d ;\nnparams= %d ;\nelitepart= %.2f ;\n"
       "sexualpart= %.2f ;\nmutprob= %.2f ;\nmutsdev= %.2f ;\n"
       "population=[\n",
       _size, _genotypeSize, _elitePart, _sexualPart,
       Genotype::getMutationProbability(), Genotype::getMutationDeviation());

  for (int i = 0; i < _size; i++)
    _genotypes[i]->save(file);

  fprintf(file, "];\n");
}

static int compare(const void *a, const void *b) {
  return (*(Genotype**)a)->getFitness() >
    (*(Genotype**)b)->getFitness() ? -1 : +1;
}

const Genotype *Population::selectParent() const {
  while (true) {
    int index = Random::getInteger(_size);
    if (index <= Random::getInteger(_size))
      return _genotypes[index];
  }
}

void Population::sort() {
  // sort for rank selection
  qsort(_genotypes, _size, sizeof(Genotype*), &compare);

  for (int k = 0; k < _size; k++)
    cout << _genotypes[k]->getFitness() << " ";

  cout << endl;

  _bestFitness[_generation] = _genotypes[0]->getFitness();
```

```
  float sum = 0.0;
  for (int i = 0; i < _size; i++)
    sum += _genotypes[i]->getFitness();

  _meanFitness[_generation] = sum / _size;
}

void Population::reproduce() {
  Genotype **_nextGeneration = new Genotype*[_size];
  for (int i = 0; i < _size; i++) {
    Genotype *child = new Genotype(_genotypeSize);

    if (i < _elitePart * _size)
      *child = *_genotypes[i];    // cloned elite
    else {
      const Genotype *mom = selectParent();
      if (Random::getUniform() < _sexualPart) {
     const Genotype *dad;
     do {
       dad = selectParent();
     }
     while (dad == mom);
     *child = mom->crossover(*dad); // sexual reproduction
      }
      else {
     *child = *mom;  // asexual reproduction
     child->singleLocusMutate();
      }
    }

    child->setFitness(0.0);
    _nextGeneration[i] = child;
  }

  for (int j = 0; j < _size; j++)
    delete _genotypes[j];

  delete [] _genotypes;

  _genotypes = _nextGeneration;
  _generation++;
}
```

6.4.20 Random.h

```
#ifndef Random_H
#define Random_H

/*
  Random number functions
  Implementation of getGaussian()
  taken over from Dr. Everett F. Carter Jr.
  http://www.taygeta.com/random/gaussian.htm
*/

class Random {
 public:
```

```
  // return random number between [0;1] from a uniform
  // distribution
  static float getUniform();

  // return random integer number between [0;max-1] from a
  // uniform distribution
  static int getInteger(int max);

  // return random number from a Gaussian distribution with
  // mean 0 and standard deviation 1
  static float getGaussian();

 private:
  Random() {} // disabled constructor
};

#endif
```

6.4.21 Random.cpp

```
#include "Random.h"
#include <math.h>
#include <stdlib.h>

const float PI = 3.1415926535f;

float Random::getUniform() {
  return (float)rand() / (float)RAND_MAX;
}

float Random::getGaussian() {
  static bool flag = true;
  static float y2;

  if (flag) {
    float x1, x2, w;
    do {
      x1 = 2.0f * getUniform() - 1.0f;
      x2 = 2.0f * getUniform() - 1.0f;
      w = x1 * x1 + x2 * x2;
    } while (w >= 1.0f);

    w = (float)sqrt((-2.0f * (float)log(w)) / w);
    float y1 = x1 * w;
    y2 = x2 * w;

    flag = false;

    return y1;
  }

  flag = true;

  return y2;
}

int Random::getInteger(int max) {
```

```
  return rand() % max;
}
```

6.4.22 RandomSearch.h

```
#ifndef RandomSearch_H
#define RandomSearch_H

/*
  Random search for locomotion parameters for modular robot
  Author: Yvan Bourquin
*/

class Robot;

#include "Optimizer.h"

class RandomSearch : public Optimizer {
 public:
  // constructor: create random search
  RandomSearch(Robot *robot);

  // destructor
  virtual ~RandomSearch();

  // run random search
  virtual void run();

 private:
  float *_params;     // current best solution
  char _filename[64]; // data file
  float _fitness;     // current solution fitness

  void save(const char *filename) const;
};

#endif
```

6.4.23 RandomSearch.cpp

```
#include "RandomSearch.h"
#include "Random.h"
#include "Robot.h"
#include <iostream>

using namespace std;

RandomSearch::RandomSearch(Robot *robot)
  : Optimizer(robot)  {

  _params = new float[getNumParams()];
  sprintf(_filename, "../../results/rs_%s%d.m", robot->getName(),
getNumParams());
  cout << "filename: " << _filename << endl;
}

RandomSearch::~RandomSearch() {
```

```
  delete [] _params;
}

void RandomSearch::save(const char *filename) const {
  FILE *file = fopen(filename, "w");
  if (! file) {
    printf("could not write file: %s\n", filename);
    return;
  }

  cout << "writing file: " << filename << endl;

  Optimizer::save(file);

  fclose(file);
}

void RandomSearch::run() {

  _fitness = 0.0;

  cout << "starting random search ... \n";

  while (! isFinished()) {

    cout << "trial: " << getNumTrials()
      << " fitness: " << _fitness << endl;

    // generate new random controller
    float *candidate = new float[getNumParams()];
    for (int j = 0; j < getNumParams(); j++)
      candidate[j] = Random::getUniform();

    float candidateFitness = Optimizer::evaluate(candidate);

    if (candidateFitness > _fitness) {
      cout << "***ACCEPTED*** candidate with fitness: "
        << candidateFitness << endl;
      _fitness = candidateFitness;
      delete [] _params;
      _params = candidate;
      Optimizer::recordSolution(_fitness, _params);
    }
    else {
      cout << "refused candidate with fitness: " << candidateFitness << endl;
      delete [] candidate;
    }

    if (getNumTrials() % 100 == 0)
      save(_filename);
  }
}
```

6.4.24 Robot.h

```
#ifndef Robot_H
#define Robot_H
```

```
/*
   Modular robot implementation for Webots simulator
   Author: Yvan Bourquin
*/

#include <device/gps.h>

class Module;

class Robot {
 public:
  // constructor: create robot according to specification
  Robot(const char *path);

  // destructor
  virtual ~Robot();

  const char *getName() const { return _name; }
  int getNumParams() const;
  int getNumModules() const { return _numModules; }

  // evaluation trial, returns fitness
  float evaluate(const float *params);

  // return true is the robot oscillators are chaotic
  bool isOver() const { return _over; }

 private:
  int _maxSteps;       // duration of a simulation trial
  const char *_name;   // robot name
  int _numModules;     // number of modules
  int _numConnections; // number of oscillator connections
  Module **_modules;   // robot's modules
  DeviceTag _gps;      // for measuring the current position
  DeviceTag _emitter;  // emitter device for communicating with supervisor
  float *_sendBuffer;  // buffer for communicating with supervisor
  bool _over;          // is this evaluation over
  int _simulationStep; // simulation step size in milliseconds
  float _x;            // current x position
  float _z;            // current z position
  float _distance;     // integrated distance
  int _stepCount;      // number of simulation steps so far

  const float *reset(const float params[]);
  float getFitness() const;
  float getAbsoluteDistance() const;
  float getIntegratedDistance() const;
  void step();
};

#endif
```

6.4.25 Robot.cpp

```
#include "Robot.h"
#include "Module.h"
```

```
#include "Defaults.h"
#include <device/robot.h>
#include <device/custom_robot.h>
#include <device/emitter.h>
#include <device/gps.h>
#include <string>
#include <iostream>
#include <cassert>

using namespace std;

Robot::Robot(const char *path) {

  _maxSteps = Defaults::get("simulation.max.steps", 500);
  _name = path;
  _over = false;
  _simulationStep = Defaults::get("simulation.step.duration", 64);
  _stepCount = 0;
  _x = 0.0;
  _z = 0.0;
  _distance = 0;
  _numModules = Defaults::get("%s.servos.count", 0, path);
  assert(_numModules > 0);
  _modules = new Module*[_numModules];

  // create one module for each servo
  for (int i = 0; i < _numModules; i++)
    _modules[i] = new Module(_name, i, _numModules);

  // create oscillator connections
  _numConnections = Defaults::get("%s.connections.count", -1, path);
  if (_numConnections < 0) {
    cerr << "ERROR: number of connections not specified.\n";
    exit(0);
  }

  for (int i = 0; i < _numConnections; i++) {
    int from = Defaults::get("%s.connection[%d].from", -1, path, i);
    int to   = Defaults::get("%s.connection[%d].to",   -1, path, i);
    if (from < 0 || to < 0)
      cerr << "ERROR: connection " << i << " is badly configured.\n";
    else
      _modules[from]->addInput(_modules[to]);
  }

  // set up emitter and gps
  _emitter = robot_get_device("emitter");
  _gps = robot_get_device("gps");
  gps_enable(_gps, _simulationStep);
  cout << "emitter device tag is " << (int)_emitter << endl;
  cout << "gps device tag is " << (int)_gps << endl;
}

Robot::~Robot() {
  for (int i = 0; i < _numModules; i++)
    delete _modules[i];
```

```
  delete [] _modules;
}

int Robot::getNumParams() const {
  return 2 * _numConnections + 2 * _numModules;
}

const float *Robot::reset(const float *params) {

  _over = false;
  _stepCount = 0;
  _x = 0.0;
  _z = 0.0;
  _distance = 0.0;

  // send message to supervisor so that he moves
  // the robot to its initial position
  cout << "sending reset message ..." << flush;
  _sendBuffer = (float*)emitter_get_buffer(_emitter);
  _sendBuffer[0] = 0;
  _sendBuffer[1] = 0;
  _sendBuffer[2] = 0;
  emitter_send(_emitter, 3 * sizeof(float));

  // make sure the supervisor has time to reset before we continue ...
  robot_step(2 *_simulationStep);

  // reset modules also
  for (int i = 0; i < _numModules; i++)
    params = _modules[i]->reset(params);

  return params;
}

// straight distance between initial and final position
// assuming that initial position was [0, ?, 0]
float Robot::getAbsoluteDistance() const {

  if (_over)
    return 0.0;

  const float *matrix = gps_get_matrix(_gps);
  float dx = gps_position_x(matrix);
  float dz = gps_position_z(matrix);

  return sqrt(dx * dx + dz * dz);
}


float Robot::getIntegratedDistance() const  {
  if (_over)
    return 0.0;

  return _distance;
}

float Robot::getFitness() const {
```

```
  float ad = getAbsoluteDistance();
  float id = getIntegratedDistance();
  float fitness = ad + id;

  cout << "absolute distance: " << ad << " integrated distance: "
       << id << " tfitness: " << fitness << endl;

  return fitness;
}

void Robot::step() {
  // simulater step
  robot_step(_simulationStep);

  // uptdate module positions
  for (int i = 0; i < _numModules; i++) {
    _modules[i]->step();
    if (_modules[i]->isCrazy())
      _over = true;
  }

  _stepCount++;

  // every second
  if (_stepCount % (1000 / _simulationStep) == 0) {
    const float *matrix = gps_get_matrix(_gps);
    float x = gps_position_x(matrix);
    float z = gps_position_z(matrix);
    float dx = x - _x;
    float dz = z - _z;
    _distance += sqrt(dx * dx + dz * dz);
    _x = x;
    _z = z;
  }
}

// evaluate robot performance during _maxStep steps
// and return fitness result
float Robot::evaluate(const float *params) {

  const float *p = reset(params);
  assert(p - params == getNumParams()); // self-test

  for (int j = 0; j < _maxSteps && ! isOver(); j++)
    step();

  return getFitness();
}
```

6.4.26 SimulatedAnnealing.h

```
#ifndef SimulatedAnnealing_H
#define SimulatedAnnealing_H

/*
   Simulated Annealing for optimizing robot locomotion
   Author: Yvan Bourquin
```

```
*/

#include "Optimizer.h"

class Robot;
class Genotype;

class SimulatedAnnealing : public Optimizer {
 public:
  // constructor: create simulated annealing for robot
  SimulatedAnnealing(Robot *robot);

  // destructor
  virtual ~SimulatedAnnealing();

  // run simulated annealing
  virtual void run();

 private:
  float _initialTemperature; // initial temperature
  float _reductionFactor;    // temperature reduction factor
  int _requiredSuccesses;    // num successes before temperature reduction
  float _temperature;        // current temperature
  float _fitness;            // fitness of current parameter set
  Genotype *_params;         // current parameter set
  int _temperatureIndex;     // temperature index
  int _nsucc;                // num successes so far for current temperature
  char _filename[64];        // data file
  Genotype **_cooling;       // cooling record

  bool metropolis(float de, float t) const;
  void save(const char *filename) const;
};

#endif
```

6.4.27 SimulatedAnnealing.cpp

```
#include "SimulatedAnnealing.h"
#include "Robot.h"
#include "Random.h"
#include "Defaults.h"
#include "Genotype.h"
#include <cmath>
#include <cassert>
#include <iostream>

using namespace std;

static const int MAX_TEMPERATURES = 1000;

SimulatedAnnealing::SimulatedAnnealing(Robot *robot)
  : Optimizer(robot) {

  _reductionFactor = Defaults::get("temperature.reduction.factor", 0.95f);
  _requiredSuccesses = Defaults::get("temperature.required.successes", 5);
  _initialTemperature = Defaults::get("temperature.initial", 1.0f);
```

```
  _params = new Genotype(getNumParams());
  _cooling = new (Genotype*)[MAX_TEMPERATURES];
  _nsucc = 0;

  sprintf(_filename, "../../results/sa_%d_%s%d.m",
       _requiredSuccesses, robot->getName(), getNumParams());

  cout << "filename: " << _filename << endl;
}

SimulatedAnnealing::~SimulatedAnnealing() {
  delete _params;

  for (int i = 0; i < MAX_TEMPERATURES; i++)
    delete [] _cooling[i];

  delete [] _cooling;
}

bool SimulatedAnnealing::metropolis(float cost, float temperature) const {
  return cost < 0.0 || Random::getUniform() < exp(-cost / temperature);
}

void SimulatedAnnealing::save(const char *filename) const {
  FILE *file = fopen(filename, "w");
  if (! file) {
    printf("could not write file: %s\n", filename);
    return;
  }

  cout << "writing file: " << filename << endl;

  Optimizer::save(file);

  fprintf(file, "initemp= %.2f ;\nredfact= %.3f ;\nmutprob= %.2f ;\nmutdev= %.2f
;\n"
       "temp= %f ;\nnsucc= %d ;\ntindex= %d ;\ncooling=[\n",
       _initialTemperature, _reductionFactor,
Genotype::getMutationProbability(),
       Genotype::getMutationDeviation(), _temperature, _nsucc,
_temperatureIndex);

  for (int i = 0; i < _temperatureIndex; i++)
    _cooling[i]->save(file);

  fprintf(file, "];\n");
  fclose(file);
}

void SimulatedAnnealing::run() {

  // initialize
  _temperatureIndex = 0;
  _temperature = _initialTemperature;
  _nsucc = 0;

  while (_temperatureIndex < MAX_TEMPERATURES && ! isFinished()) {
```

```
    while (_nsucc < _requiredSuccesses && ! isFinished()) {

      // generate new configuration
      Genotype *candidate = new Genotype(*_params);
      candidate->singleLocusMutate();

      float candidateFitness = Optimizer::evaluate(candidate->getGenes());
      float cost = _fitness - candidateFitness;

      cout << "trial: " << getNumTrials()
        << " temperature: " << _temperature
        << " fitness: " << _fitness
        << " nsucc: " << _nsucc << endl;

      if (metropolis(cost, _temperature)) {
     delete _params;
     _params = candidate;
     _fitness = candidateFitness;
     _params->setFitness(_fitness);
     Optimizer::recordSolution(_fitness, _params->getGenes());
     _nsucc++;

     cout << "***ACCEPTED*** candidate with fitness: "
          << candidateFitness << " cost: "<< cost << endl;
      }
      else {
     cout << "refused candidate with fitness: "
          << candidateFitness << " cost:"<< cost << endl;

     delete candidate;
      }

     if (getNumTrials() % 100 == 0)
     save(_filename);
    }

    cout << "***REDUCING TEMPERATURE*** ..." << endl;

    _cooling[_temperatureIndex] = new Genotype(*_params);
    _temperature *= _reductionFactor;
    _temperatureIndex++;
    _nsucc = 0;
  }
}
```

6.5 C++ listings of supervisor controller

```
#include "Defaults.h"
#include <device/robot.h>
#include <device/supervisor.h>
#include <device/receiver.h>
#include <stdio.h>
#include <iostream>

using namespace std;

static int numServos = -1;
```

```
static DeviceTag receiver;
static float (*servo_current_positions)[7];
static float robot_initial_position[7] = { 0, 0, 0, 0, 0, 1, 0 };
static float (*servo_initial_positions)[7];
static NodeRef servos[99];
static NodeRef robot;

void reset(void) {
  receiver = robot_get_device("receiver");
  cout << "supervisor: receiver device tag is " << (int)receiver << endl;
}

int run(int ms) {
  int length = receiver_get_buffer_size(receiver);
  if (length) {
    (float*)receiver_get_buffer(receiver);

    cout << "supervisor: resetting ..." << endl;

    supervisor_simulation_physics_reset();

    supervisor_field_set(robot,
               SUPERVISOR_FIELD_TRANSLATION_AND_ROTATION,
               robot_initial_position);

    for (int i = 0; i < numServos; i++) {
      supervisor_field_set(servos[i],
                 SUPERVISOR_FIELD_TRANSLATION_AND_ROTATION,
                 servo_initial_positions[i]);
    }

    supervisor_simulation_physics_reset();
  }

  return 64;
}

int main(int argc, char * argv[]) {

  Defaults::loadFile("../walker.ini");
  const char *robotName = Defaults::get("robot.name", (const char*)NULL);
  if (! robotName) {
    cerr << "supervisor: error: robot name not specified !\n";
    return 0;
  }

  numServos = Defaults::get("%s.servos.count", -1, robotName);
  if (numServos < 0) {
    cerr << "supervisor: error: number of servos not specified !\n";
    return 0;
  }

  robot_initial_position[6] =
    Defaults::get("%s.start.z.rotation", 0.0, robotName);

  robot_live(reset);
  receiver_enable(receiver, 64);
```

```
  robot = supervisor_node_get_from_def("WALKER");

  for (int i = 0; i < numServos; i++) {
    const char *servoName = Defaults::get("%s.servo[%d].name",
                            (const char*)NULL, robotName, i);
    cout << "supervisor: looking for " << servoName << " ... ";
    servos[i] = supervisor_node_get_from_def(servoName);
    if (servos[i])
      cout << "found" << endl;
    else {
      cout << "NOT FOUND !" << endl;
      break;
    }
  }

  servo_current_positions = new float [numServos][7];
  servo_initial_positions = new float [numServos][7];

  for (int i = 0; i < numServos; i++)
    supervisor_field_get(servos[i],
                SUPERVISOR_FIELD_TRANSLATION_AND_ROTATION,
                (void*)servo_current_positions[i], 128);

  robot_step(128);

  for (int i = 0; i < numServos; i++)
    for (int j = 0; j < 7; j++)
      servo_initial_positions[i][j] = servo_current_positions[i][j];

  robot_run(run);

  return 0;
}
```