Self-Organisation of Locomotion in Modular Robots: A Case Study

Bertrand Mesot

February 19, 2004

Contents

Chapter 1

Introduction

The aim of this project is to set out the ground for the future development of a self-reconfigurable modular robot. Since the ideas behind the concept of simple small robots cooperating to achieve a certain task is not new we will begin by summing up the properties of existing projects. Some concepts are however common to all projects involving the design of self-reconfigurable modular robots, being adaptive is probably the most important of them. Indeed the main problem faced by classical robotics is the lack of adaptiveness to environmental events. Since the early years of robotics this point is still the most difficult to address.

Modular robotics is a kind of antithesis to classic robotics. Instead of building one complex monolithic robot we build many simple adaptive ones. Of course those simple units alone are far from being able to cope with even the most simple task achieved by a monolithic robot. However by giving them the capabilities of interacting with each other—for example, by information transfers or active/passive links—we hope they will be able to achieve complex tasks with a high degree of robustness.

As bright as this new concept seems to be, making usable and useful robots out of it is far from being easy. The recent and rapid successes of swarm intelligence as well as the will to apply it to embedded robotics somehow open the way toward modular robotics. The later however goes a step further by adding assembling capabilities which allow a group of robots to change its shape according to the situation it is involved in. Projects making use of swarm intelligence or some kind of earlier stage modular robotics are for example [8, 13, 16] and [9, 10, 11].

This chapter first gives a broad description of world known projects that deal with modular robotics. It exposes their advantages, interesting ideas as well as the main encountered problems. An overview of the subjects treated by this document is then presented, each section will expose in a rather informal way the content of each of the following chapters.

1.1 Existing Projects

Let us now look at some world known projects related to modular robotics. Beginning with the works of Toshio Fukuda et al. back in 1990 [7], the field of modular robotics started to rise quickly. Strangely enough although a number of labs are working in the field, there is only two great classes of modular robots. The first is more "biologically-inspired" since the shape of the robots as well as the way they move remind of animals like snakes, spiders or insects. The second finds its roots in particle physics and its computer science counterpart, cellular automata. Contrary to the first class, units are simple and small and above all numerous [19].

1.1.1 The XEROX Polybot Project

The first paper setting the ground for what will become the XEROX Polybot Project was published back in 1993. The project is thus not new, but becomes famous during the last four or five years, because of the well known hardware implementation the Polybot team made of their modular robots [6, 28]. Polybot modules are made of simple units which have no way to move by themselves. However, once connected together, they are able to coordinate their actions in order to achieve locomotion on rough and complex terrains.

As we will see through this document modular robots are preferred over classical ones mainly because of their reconfiguration capability, however such a capability is not easy to achieve since docking needs nearly perfect alignment. Furthermore since computation is distributed over a great number of elements, problems of synchronisation and coordination of the various units arise. To solve this difficulties the XEROX Polybot *generation 3* (G3) uses an *Attribute/Service Model* (ASM) which allows the building of applications distributed on many thread and many processors. The overall forms a *Massively Distributed Control Network* which supports messages passing through sockets and thus provides reliable communication between various units [29, 30].

In the second generation (G2) of Polybot, locomotion was achieved by means of control tables i.e. small finite states machines that control each unit. It used a master/slave communication scheme where a defined master unit was computing the control table of every slave. The problem is that this solution is far from being scalable since as more units are added, the master quickly becomes a bottleneck. Moreover control table only act in open-loop i.e. it does not interact with the environment. To overcome those difficulties the G3 Polybot uses a *phase automata pattern* which is an event-driven discrete state machines with a periodic behaviours. For example the well known loop gaits is achieved by a four states machine and may easily takes into account events arising from the environment by the addition of what is called *conforming* states. Once in a conforming state a module signals to the others that it was not able to take the right action because of an external event. Other modules are thus consequently able to adapt their behaviour [27].

Polybot reconfiguration is achieved through *constraint solvers* which make use of inverse kinematics in order to precisely align two modules. Constraint solvers are distributed programs which take raw data coming from IR sensors

and accelerometers in order to correctly control joint motors and thus achieve precise docking for example [26, 30].

1.1.2 The CONRO Project

The aim of this project is to build "[...] small, self-sufficient and relatively homogeneous" modules which are able to "[...] reassemble at will". Those robots are "[...] designed for real-world application such as fire fighting, urban and rescue after an earthquake, and battlefield reconnaissance." A detailed explanation of what CONRO module is made of can be found in [25]. Primarily CONRO's module are considered as a small independent unit, thus possessing its own sensors, actuators and CPU, as well as communication capabilities.

Since the main focus is real-world applications, a thorough study of hardware issues has been carried out and two physical prototypes were build. Those experiments bring the conclusions that a single module must be able to lift several other modules, thus it must be small enough in order to minimise power consumption while allowing a sufficiently large battery to be carried. Moreover it has been found that "[...] the most challenging task is probably the docking control". Indeed since the overall behaviour depends on the way modules assemble themselves, docking should be sufficiently changeable to permit to easily grab and release modules while allowing inter-module communication, power exchange and precise control of rotation axis.

Robot control is achieved through local (intra-module) or wireless (inter-robot) communications and may work in a master/slave or distributed mode. When acting in the later mode an hormone-based control algorithm is used. This algorithm manage inter-module synchronisation, action specialisation and also controls shape reconfiguration [4, 20, 21, 22]. Locomotion is also accomplished by means of hormones which define actions performed by each module and carry synchronisation signals. Each modules is provided with two motors controlling two rotation axis. Locomotion is thus achieved by changing the angular position of each motor according to a simple algorithm.

1.1.3 The Molecule and Crystal Robots

The Molecule is the first prototype of a modular robot built at the Dartmouth Robotic Lab. It prefigures the transition form "classical" modular robot i.e. an assembly of small modules that looks like or moves like an animal, to more regular and compact structures that are inspired by cellular automata. Indeed reconfiguration is the only capability of a Molecule robot, locomotion is thus achieve thanks to it and is not, like in the XEROX Polybot or CONRO robot, only a way to be adaptive [19].

The Dartmouth team goes one step further with what they call the Crystal robot. The Crystal robot is inspired by biological muscles, it is composed of Crystalline atoms which are very simple units that can attach themselves to their relatives and use a expansion/contraction mechanism to move relatively to their neighbours. Similarly to the Molecule robot, locomotion is achieve through reconfiguration of the whole robot only. However the great number of

Crystalline atom that form a Crystal robot requires the development of robust and efficient distributed algorithm. One of them, called *PACMAN*, operates by placing "pellets" along the path a given atom should follow. However a single atom never move along the while path, but instead exchange its identity with its neighbours. The overall effect resulting of the local activity of many atoms is that the whole robot reconfigure itself, move and even duplicate itself [2, 3, 19].

1.2 Overview

It is obvious when looking at existing projects that a number of problems arise from the use of modular instead of monolithic robots. Since those difficulties are hardly addressable as a whole we should concentrate on only a small subset of them. Indeed common constraints like power consumption, sensors/actuators integration, electrical wiring, docking properties and joints design are highly dependant on current available technologies. We may thus reasonably suppose that one day chips, motors and batteries will be small enough to fit our needs—some researchers are already thinking about nano-scale robots.

Although hardware gets better with time, models and physical law should not change that much. This is the reason why in this document, we will voluntarily occult all hardware related problems and concentrate only on how to achieve self-organisation of a group of connected homogeneous and simple modular robots. To stay as general as possible we shall make only some little assumptions on the design of what we will call a module. Physical simulations will then allow us to easily test our theories and qualitatively measure their relevance when confronted to the real world.

1.2.1 The Module

Our module is made of two parts with two additional sticks-wheels plugged on the longest one. The two parts are connected by an universal joint, providing two degrees of freedom. The long part represents two third of the total length and possess three docking points situated on the top, bottom and rear side. Similarly the small part possesses five docking points situated on each of its free sides. Figure 1.1 shows the design we choose to give to the module. Since physical simulations of at least twenty modules is still a rather time consuming process we choose to limit the possible primitives used to modelise the module to only boxes. Moreover the joint linking the two parts is considered dimensionless, consequently contacts between the two parts are not taken into account. In order to conserve a physical meaning, joint's angle are however limited to a certain range of values.

1.2.2 Docking

One of the main problem of modular robotics is the design of the joint which glues modules together. Although we do not deal with physical constraints relative to the design of those joints, we still need to model their behaviour. Docking

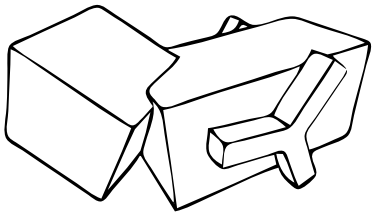


Figure 1.1: Our module.

will thus occur between two modules if they are close enough and their common face are well aligned. Repetitively checking for a possible docking situation during simulation takes $\mathcal{O}(n^2)$ computations, we will thus instead suppose that docking may only arise if two modules collide. Since contact points are automatically computed by the physical simulator, only an additional calculation will be carried out in order to compute faces alignment. If all conditions are fulfilled then docking will occur and a joint will be added between the two concerned modules.

We should keep in mind that while removing physical constraints, simulations introduce new problems. Indeed by linking two modules, one introduce strong physical constraints on the system. In the real world those constraints are smoothly integrated through friction, in simulation however they are instantaneously added which may cause critical instabilities, docking thus remains a critical problem we need to address.

1.2.3 Brownian Motion

In existing project like CONRO or XEROX Polybot, possible modules configurations were set by hand. Indeed reconfiguration is a far from simple process since it relies on the precise docking of many modules. For example transforming a snake-like structure into a loop needs a certain bend which arises from the precise setting of each joint's angle. Since our modules are provided with sticks-wheels we will instead use them to randomly assemble robots which will then exclusively use their joint motors to move.

In a first phase, modules will simply wander in kind of brownian motion while being open to any potential linking with one of their similar. Moreover joints motors will be set in a passive mode thus allowing module's smaller part to move freely. If the area available is finite, like a circle arena for example, group of modules will form. Once connected a module will stop its wheels and only use its joint motors. This state of the system is a kind of transient phase where connected modules will cohabit with free ones. Then, maybe two groups will collide thus giving birth to a bigger group. Since bigger groups are harder to avoid, then sooner of later all modules will be connected to the same group.

1.2.4 Locomotion

The first goal of a group of interconnected modules is to find the optimal way to achieve locomotion. Since sticks-wheels will not be used anymore at this stage, robots should only count on their joint motors in order to move. Locomotion is obviously a global task which need a precise organisation of each part of the system. Since modules are randomly connected, we thus need a way to find (or define) the actions that each module need to carry out in order to participate in the right way to the overall behaviour.

1.2.5 Reconfiguration

The main interest for modular robotics is probably directed toward reconfiguration. Indeed reconfiguration capabilities was extensively studied in the CONRO and XEROX Polybot projects for example. Moreover in the Crystalline or Molecule robots, reconfiguration is their unique capability and locomotion is thus achieve through it. Reconfiguration is obviously a mandatory issue to address when dealing with modular things, however we will not cover it in this document for two reasons. The first is that, as previously mentioned, although reconfiguration has been well studied, self-organised locomotion remains an open problem that need to be addressed. The second reason is that reconfiguration of non-lattice structure is far from simple as can be attested by the numerous papers dealing with this subject. Since at that time no generic and good enough solution has been found, addressing this problem will largely go beyond the realm of this project. Moreover due to its high complexity and dependence to physical constraints, reconfiguration is certainly a problem that should not be investigated through simulations only.

Chapter 2

In-Depth Description

This chapter provides an in-depth description of the module's structure as well as how brownian motion and docking is achieved. It deals exclusively with everything that a module does before it get connected to one of its similar. It also presents the assumptions made on motors' power and size of modules. Of course what may be simulated tightly depends on the capabilities of the simulator, we shall thus begin by a brief description of the one we used.

2.1 Open Dynamics Engine

All physical simulations done during this project will use the Open Dynamics Engine, abbreviated ODE. ODE author says that "ODE is a free, industrial quality library for simulating articulated rigid body dynamics—for example ground vehicles, legged creatures, and moving objects in VR environments. It is fast, flexible, robust and platform independent, with advanced joints, contact with friction, and built-in collision detection." It thus provides all the basic tools we need in order to correctly simulate the behaviour of real modules and their interactions with the world.

2.2 The Module

Our module is 6 cm long and composed of two parts of $4 \times 2 \times 2\,\mathrm{cm}^3$ and $2 \times 2 \times 2\,\mathrm{cm}^3$, the density was chosen to be around $1\,\mathrm{g/cm}^3$ because of possible underwater applications. The total mass is thus 24 g which may seems underestimated given current technologies, but since we do not want to bother ourself with hardware limitations we shall suppose that in a near future the needed technology will be available and we shall thus concentrate on open problems.

2.2.1 Angular Motors

Module's joint is controlled by two motors, one for each of the two rotation axis. Of course motors are limited in the range of angles they can reach. For

convenience we will choose to limit angles between $-\pi/2$ and $\pi/2$ for both motors i.e. the small part of the module may be inclined in this range, relative to the long part (see Figure 1.1).

ODE motors implementation allows the control of angular velocity only, consequently angular control should be transposed into angular velocity control. A basic and straight forward way to do this transformation is:

$$\dot{\vartheta}_{t+1} = \kappa_p(\hat{\vartheta} - \vartheta_t)$$

where $\hat{\vartheta}$ is the desired angle, ϑ_t is the current angle at time t and κ_p a problem specific constant to be determined. However one additional issue with real motors is the fact that they do not achieve instantaneous motion i.e. time is required for a motor to reach a desired angle, if it goes too fast it may exceed the asked angle, going slower resolve the problem, but more time is needed. This is a classical problem in control theory and is commonly solved by using a PD controller which has the following form:

$$\dot{\vartheta}_{t+1} = \kappa_p(\hat{\vartheta} - \vartheta_t) + \kappa_d\dot{\vartheta}_t$$

where κ_d is a problem specific constant to be determined. Finding well suited κ_p and κ_d must be done by hand since they are highly dependant on motors properties, the mass to put in motion and the degree of precision we are looking for. However more or less automatic methods based on experimental results exist, for example Ziegler-Nichols method, but since in our case, modules are randomly connected κ_p and κ_d should always be adjusted. Fortunately ODE already provide joint error correction in the form of an *error reduction parameter* (ERP), the level of correction is defined by a parameter whose default value appears to work just fine for us. Furthermore, after some tests, ERP proved to be able to compensate the need for κ_d.

The main constraint an angular motor should fulfil is to be able to lift at least three modules. In order to prevent possible limitations due to a lack of power, simulated motors will provide a torque of 5 Nm which allows, when $\kappa_p = 40$, to easily and precisely lift five modules against gravity in about one tenth of second.

2.2.2 Docking

Mechanical implementation of docking is probably one of the most complex tasks one is faced when building real world robots. Thanks to the use of simulation it becomes tractable, but remains however a tricky part. Indeed in a simulator like ODE, on one hand simple objects may be composed to form more complex ones and on the other hand composite or simple objects may be linked by joints, but composed objects may not be build during simulation, whereas joints may be created at any time. Six types of joint exist, ball, hinge, hinge2, universal and fixed. The last one should however not be used although it would be convenient in our case. The reason is that the instantaneous creation of a joint between two bodies is bad for the stability of the simulated system, because it creates hard physical constraints which may require large forces to be satisfied, and those forces may break the whole system down. Since all other ODE joints make

use of ERP and it is possible to set their low/high stop angles as well as their maximum torque, simulating a fixed joint is easily an robustly done by using an universal or hinge joint where low and high stop angles are set to small values (-0.01 and 0.01 in our case) and the maximum torque to a sufficiently large (50 Nm in our case) value to prevent any parasitic movements to occur.

ODE provides a way to be aware of any collisions between two bodies, this is of great help since two colliding bodies may require docking. Three tests will then be carried out in order to determine whether or not a link is created. Let's call S_1, S_2, $\mathbf{n}_1$, $\mathbf{n}_2$, $\mathbf{c}_1$ and $\mathbf{c}_2$ the two colliding surfaces and their respective normals and centre, then the three tests become:

1. Let's define $\mathbf{p} = \mathbf{c}_2 - \mathbf{c}_1$ and let's $d = \mathbf{n}_1^T \cdot \mathbf{p}$ be the length of the projection of the vector joining the centre of each face. It gives how far away the two colliding surfaces lay from each other. If $d \leqslant d_{\max}$ then the test is passed. Of course since we wait for collision before doing the tests, d should always be sufficiently small. However this test prevents docking of two bodies that are too far away after a collision, in case of an elastic shock for example.

2. Let's define $\gamma = \cos^{-1}(d/\|\mathbf{p}\|)$ and let's $l = \|\mathbf{p}\| \sin(\gamma)$ be the length that separate $\mathbf{c}_1$ from the projection of $\mathbf{c}_2$ on S_1. If $l \leqslant l_{\max}$ then the test is passed. This measure gives an idea of how well $\mathbf{c}_1$ and $\mathbf{c}_2$ are aligned. The best alignment occurs when $\mathbf{p}$ is parallel to $\mathbf{n}_1$.

3. Let's $\delta = \cos^{-1}(\mathbf{n}_1^T \cdot \mathbf{n}_2)$ be the angle between $\mathbf{n}_1$ and $\mathbf{n}_2$. If $|\delta - \pi| < \delta_{\max}$ then the test is passed. This measure give information on how well S_1 and S_2 are aligned by comparing their normals. The best situation arises when $\delta = \pi$ since S_1 and S_2 face each other.

By influencing $d_{\max}$, $l_{\max}$ and $\delta_{\max}$ we are now able to control docking abilities of our modules. For example, setting $l_{\max}$ to 1 cm will allow the centre of S_2 to be situated into a circle of radius 1 cm around $\mathbf{c}_1$. Furthermore our module possesses eight anchor points, five of them are on the small part—one anchor by free face—and the remaining three are situated on the top, bottom an back of the large part. For practical reasons no anchor is placed on the legs.

We now possess everything that is needed in order to correctly simulate docking between our modules. One important point however still remains, how should we set $d_{\max}$, $l_{\max}$ and $\delta_{\max}$? Obviously those parameters highly influence the quality of docking, on one hand if they are too small no docking will occur or it will take a long time, on the other hand if they are too slack docking will occur too easily. What we need is a good compromise between precision and speed. Let's thus define ranges of values that each parameter is allowed to vary in, and let's measure, after a finite time interval, the following things:

1. The number of connected components i.e. the number of groups of connected modules—group of only one module (singleton) are also considered,

2. The mean size of those groups,

3. The mean number of connections per module in those groups.

Figure 2.1 and 2.2 shows how those quantities vary for different values of $l_{\max}$ and $\delta_{\max}$ when $d_{\max} = 0.24\,\text{cm}$ and $0.5\,\text{cm}$ respectively. The curves were obtained by averaging the results of 50 tests which simulated 25 modules during 300 seconds for each combination parameters in the following sets:

- $d_{\max} \in \{0.24, 0.5\}\,\text{cm}$,
- $l_{\max} \in \{0.2, 0.4, 0.6, 0.8, 1\}\,\text{cm}$,
- $\delta_{\max} \in \{2, 4, 6, 8, 15, 20\}$ degrees.

Those experiments lead to the following observations:

1. $d_{\max}$ plays no influence on the way modules attach together. This is consistent with the fact that we wait for collision between two modules before testing if docking may arise.
2. Small values of $l_{\max}$ create more groups which are composed of a small number of elements with a low number of connections per modules.
3. The number of connected components decreases as $\delta_{\max}$ increases. Of course the size of the groups and the mean number of connections per modules increases.

For the remaining experiments we will choose the value of each parameter as follow: $d_{\max}$ does not influence docking, thus taking the smallest value seems a good bet, from Figure 2.1 we see that $l_{\max} = 0.4\,\text{cm}$ is just the middle case between too tight and too loose constraints, it thus seems appropriate, finally a value of 7 degrees for $\delta_{\max}$ seems right since it seems to be also a good compromise i.e. the point corresponding to $\delta_{\max} = 7$ is roughly situated between the two extrema of each curve.

2.2.3 Brownian Motion

We will use the term "brownian motion" in its generic sense here to characterise the random motion of our modules in their environment. Since modules move freely and randomly in a limited area, sooner or later they will collide with other ones, maybe anchoring and thus building arbitrary shaped groups. In fact the groups' shape is not so random, because modules tend to attach with an higher probability to big groups—they are covering more space—groups will tend to have a structure similar to those observed in *reaction-diffusion systems*.

In order to carry out a good looking random movement, modules continuously change their speed by following a simple method based on the difference of speed between the left and right wheels. The speed of both wheels is initially set to the desired average speed of the module. After a given period of time the speed of one wheel is slightly incremented whereas the speed of the other is decremented by the same amount. Additionally in order to allow a module to go forward as well as backward equally well, after three period the speed of both wheels is inversed with a probability of 1/2. This algorithm is quite empirical but gives good results and above all prevents that all modules stay close to the arena's boundary.

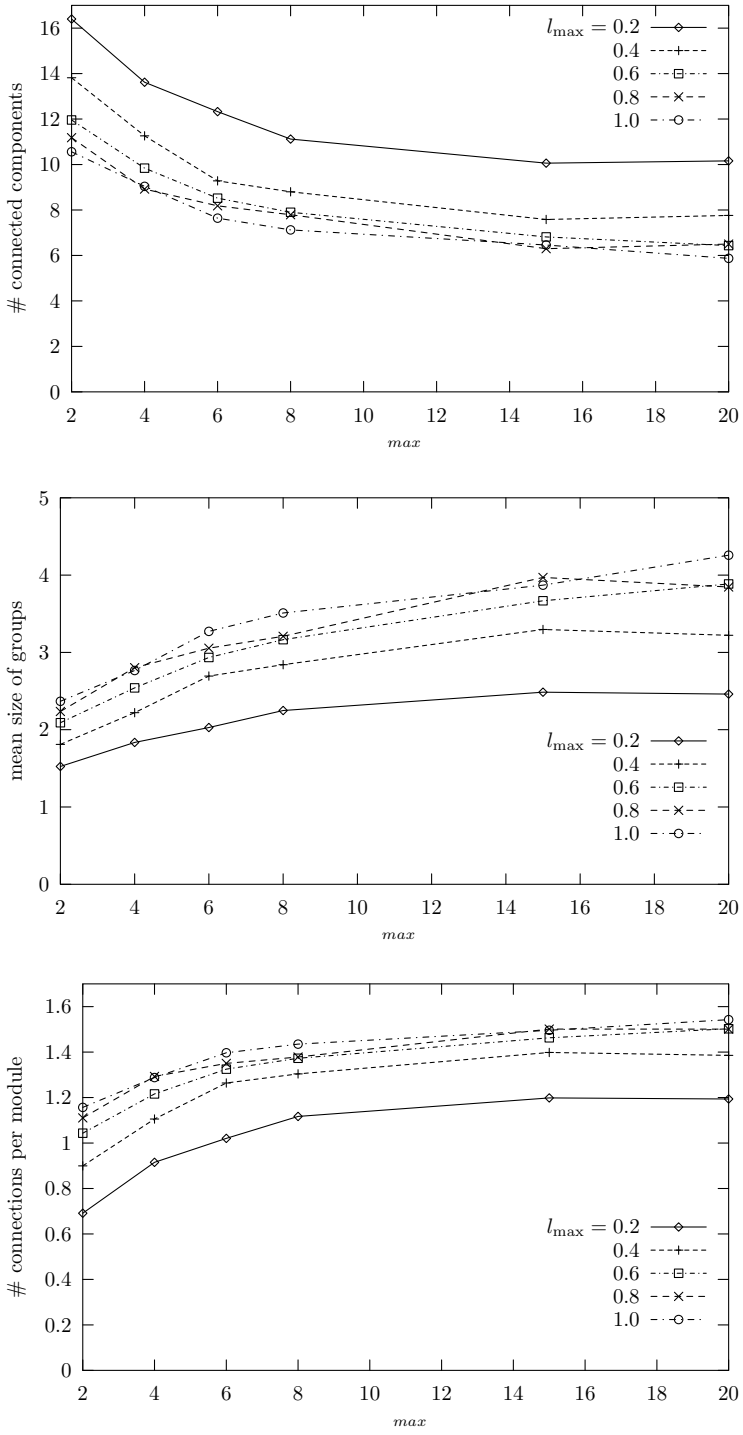


Figure 2.1: Number of connected components, mean size of groups and number of connections per module for different value of $l_{\max}$ [cm] and $\delta_{\max}$ [deg] when $d_{\max} = 0.24\,\text{cm}$.

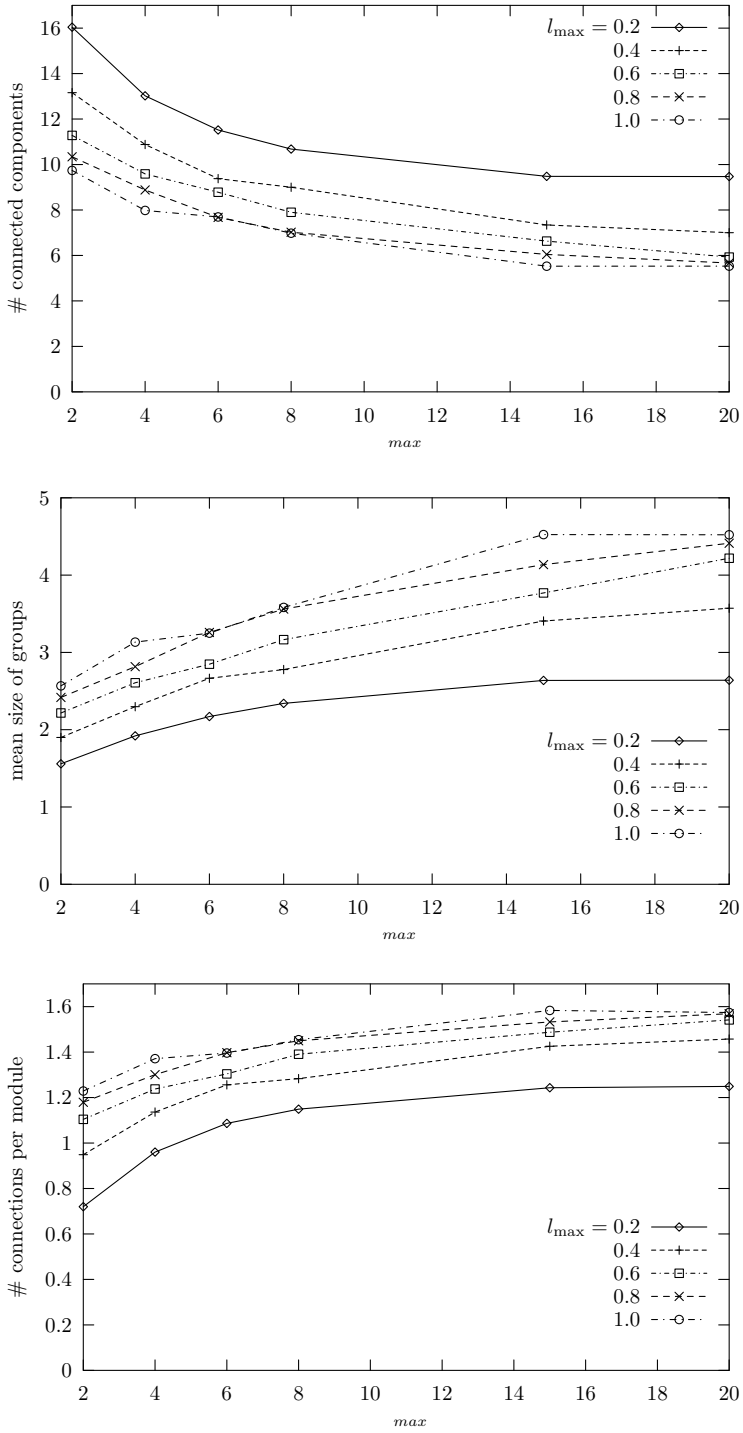


Figure 2.2: Number of connected components, mean size of groups and number of connections per module for different value of $l_{\max}$ [cm] and $\delta_{\max}$ [deg] when $d_{\max} = 0.5\,\text{cm}$.

Chapter 3

Locomotion through Oscillations

In order to achieve locomotion, connected modules use their joint motors. Since oscillations provide a good framework for the generation of gaits in animals, using them to control locomotion seems a pretty good approach. Moreover instead of concentrating on how those oscillations are produced—using neural networks for example—we simply suppose that they exist. This scheme is a good compromise between an over detailed and a too sparse modelisation of the system. Indeed since we know that the biggest part of animal's locomotion implies oscillations, taking into account all the process that creates those oscillations does not seem necessary.

3.1 The Model

From now on we will consider that the two joint motors of a module are controlled by oscillators. In order to achieve complex locomotion those oscillators need to be connected and should be robust when faced to perturbations. Since it is not clear from the explicit form of the equation of a simple oscillator how those two conditions may be achieved, we will instead describe oscillations using nonlinear equations.

A simple one dimensional oscillation my be described by the following explicit equation:

$$x = A\sin(\vartheta) + \tilde{x} \quad \text{and} \quad \vartheta = 2\pi\nu t + \phi \tag{3.1}$$

where A, $\tilde{x}$, ν and ϕ are the amplitude, reference position, frequency and phase of the oscillations. An implicit differential equation may be derived from Equation 3.1 by the following transformation:

$$\begin{aligned} x &= A\sin(\omega t + \phi) + \tilde{x}, \\ \dot{x} &= A\omega\cos(\omega t + \phi), \\ \ddot{x} &= -A\omega^2\sin(\omega t + \phi) = -\omega^2(x - \tilde{x}). \end{aligned}$$

Rewriting the last line into a first order linear differential equation gives us:

$$\begin{cases} \tau\dot{x} &= v \\ \tau\dot{v} &= -(x-\tilde{x}) \end{cases} \tag{3.2}$$

which describe a generic harmonic oscillation around $\tilde{x}$. Let's note however that A is implicitly defined in Equation 3.2. Indeed since Equation 3.1 is equivalent to Equation 3.2 then $A = ((x-\tilde{x})^2 + v^2)^{1/2}$ which means that a perturbation of any of the two state variables x or y also changes the amplitude. This is obviously not a robust behaviour.

To force the system to operate at a precise amplitude a dissipative term need to be added to Equation 3.2 which then becomes nonlinear and thus harder to analyse. This term should be equal to zero when the desired amplitude is achieved and drive the system toward the right amplitude otherwise. Since we know from Equation 3.1 that $((x-\tilde{x})^2 + v^2) - A^2$ is zero for oscillations at amplitude A, it makes a good dissipative term. Adding it to Equation 3.2 gives us:

$$\begin{cases} \tau\dot{x} &= v \\ \tau\dot{v} &= -\alpha\dfrac{(x-\tilde{x})^2 + v^2 - E}{E}v - (x-\tilde{x}) \end{cases} \tag{3.3}$$

Since $(x-\tilde{x})^2 + v^2$ may be understood as the energy of an harmonic oscillator, A^2 can be renamed in E [12]. The dissipative term is also normalised in order to prevent oscillators with large amplitude to be more influenced than those with smaller ones. Furthermore multiplying by v implies that the system converges faster when the speed is high. Finally α is a constant which set how quickly the system converge toward the limit cycle, a greater α implies a quicker convergence.

The second important part of our model concerns the coupling between oscillators. Following [12] we use a simple linear coupling which depend on the position and velocity of the coupled oscillators. Equation 3.3 for a particular oscillator σ_i is then rewritten as:

$$\sigma_i : \begin{cases} \tau_i\dot{x}_i &= v_i \\ \tau_i\dot{v}_i &= -\alpha\dfrac{(x_i-\tilde{x}_i)^2 + v_i^2 - E_i}{E_i}v_i - (x_i-\tilde{x}_i) + \displaystyle\sum_j \Gamma_j \\ \Gamma_j &= a_{ij}(x_j-\tilde{x}_j) + b_{ij}v_j \end{cases} \tag{3.4}$$

A problem still remains since large variation of x_j or v_j will dominate over small ones. Moreover in the following sections we will deal with the problem of setting a precise phase difference between two oscillators, this is commonly achieved by setting a_{ij} and b_{ij} to some value. However according to Equation 3.4 if x_j or v_j is ten times bigger, a_{ij} or b_{ij} need to be ten times smaller in order to keep the same influence. Since we expect to find that the phase difference ϕ_{ij} between oscillator i and j is a function of a_{ij} and b_{ij} only, x_j and v_j need to be normalised. Equation 3.4 is thus finally rewritten in:

$$\sigma_i : \begin{cases} \tau_i\dot{x}_i &= v_i \\ \tau_i\dot{v}_i &= -\alpha\dfrac{(x_i-\tilde{x}_i)^2 + v_i^2 - E_i}{E_i}v_i - (x_i-\tilde{x}_i) + \displaystyle\sum_j \Gamma_j \\ \Gamma_j &= \left((x_j-\tilde{x}_j)^2 + v_j^2\right)^{-1/2}\left(a_{ij}(x_j-\tilde{x}_j) + b_{ij}v_j\right) \end{cases} \tag{3.5}$$

3.2 Phase Synchronisation

In the previous sections we thoroughly detailed the oscillators, let's now see how to synchronise them. Our main problem is that we have a set of interconnected oscillators and according to theories on weakly coupled nonlinear oscillators [17], if the frequency detuning is not too large then two coupled oscillators tend to synchronise i.e. the phase difference stay constant through time. Of course the phase difference is a function of the coupling parameters (a_{ij} and b_{ij} in our case), however no general enough theory is able to predict it. Such a theory however exists for some systems where amplitudes and phases are unrelated (see for example [1]) but it is not the case of Equation 3.5.

Instead of trying to derive analytical equations describing the relations between phase differences and coupling parameters, we will follow a simple error based dynamical approach, mainly because analytical solutions may be complex and somehow limited by their range of application. Furthermore those solutions may not be well adapted to gaits transition where a transient state is needed to change from one kind of locomotion to another. In this case one must explicitly provide equations describing the transition. Although this may be still feasible for constant shaped systems, it would be difficult to do in our case since we do not know anything about the way oscillators are connected.

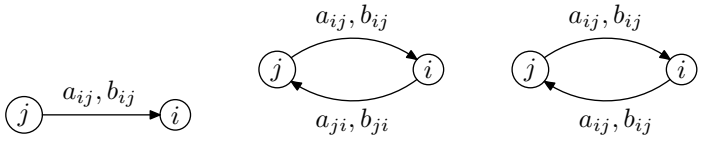
3.3 A Simple Adaptive Algorithm

Locomotion implies precise synchronisation between oscillators. Most of the time we know what are the required phase differences in order to generate a desired locomotion—for example walk, trot or bound as shown in [5]—but setting them directly is almost never possible. In the case of Equation 3.5 the phase difference is implicitly defined by the value of a_{ij} and b_{ij} and cannot be explicitly set to a given value. Since our main concern is the self-organisation of locomotion in randomly connected modules, and we know that locomotion need precise phase differences, being able to set those values directly seems to be far more convenient than having to tune a_{ij} and b_{ij}.

If perturbations are small enough and the oscillator σ_i defined by Equation 3.5 is on its limit cycle, then its behaviour is close to the sine function described by Equation 3.1. In fact if there is no coupling at all and σ_i is on its limit cycle, Equations 3.5 and 3.1 are equivalent, but when a coupling exists the equivalence is lost. We may thus see Equation 3.1 as the ideal trajectory an oscillator should follow and compare it to the real trajectory given by Equation 3.5. Of course the best case arise when the difference between the ideal and real positions is the smallest. Formally let's suppose two oscillators connected by one link as shown on Figure 3.1(a). If we call ϕ_{ij} the constant phase difference that oscillator i must keep relatively to j, then we get:

$$\hat{x}_i = A_i \sin(\vartheta_j + \phi_{ij}) + \tilde{x}_i \quad \text{and} \quad \hat{v}_i = A_i \cos(\vartheta_j + \phi_{ij}) \tag{3.6}$$

where $\hat{x}_i$ and $\hat{v}_i$ are the desired position and velocity of oscillator i. If the right phase difference ϕ_{ij} is achieved between i and j then $x_i \approx \hat{x}_i$ and $v_i \approx \hat{v}_i$. This



(a) (b) (c)

Figure 3.1: Three possible connection schemes: (a) forward only with two free parameters, (b) recurrent with four free parameters and (c) recurrent where backward parameters depend on forward ones.

lead to the following error function:

$$\begin{aligned} \mathcal{E}_i &= \tau_i \dot{v}_i - \left[-\alpha \frac{(\hat{x}_i - \tilde{x}_i)^2 + \hat{v}_i^2 - E_i}{E_i} \hat{v}_i - (\hat{x}_i - \tilde{x}_i) + \sum_j \Gamma_j \right] \\ &= \tau_i \dot{v}_i + \alpha \frac{A_i^2 - E_i}{E_i} A_i \cos(\hat{\vartheta}_i) + A_i \sin(\hat{\vartheta}_i) - \sum_j \Gamma_j \end{aligned} \tag{3.7}$$

where $\hat{\vartheta}_i = \vartheta_j + \phi_{ij}$ is the desired theta. Applying gradient descent to the quadratic error then lead to:

$$\Delta a_{ij} = -\eta \frac{\partial \mathcal{E}_i^2}{\partial a_{ij}} = 2\,\eta\,\mathcal{E}_i \frac{\partial \Gamma_j}{\partial a_{ij}} = 2\,\eta\,\mathcal{E}_i \frac{x_j}{\left((x_j - \tilde{x}_j)^2 + v_j^2\right)^{1/2}} \tag{3.8}$$

$$\Delta b_{ij} = -\eta \frac{\partial \mathcal{E}_i^2}{\partial b_{ij}} = 2\,\eta\,\mathcal{E}_i \frac{\partial \Gamma_j}{\partial b_{ij}} = 2\,\eta\,\mathcal{E}_i \frac{v_j}{\left((x_j - \tilde{x}_j)^2 + v_j^2\right)^{1/2}} \tag{3.9}$$

which show how to change a_{ij} and b_{ij} in order to reduce the total quadratic error between the desired and real state of oscillator i i.e. synchronising j and i with a phase difference of ϕ_{ij}. The parameter η plays the role of a learning constant and is commonly set to a small value (0.001 for example).

3.4 Improvements

On one hand, gradient descent is rarely used as such, since most of the time useful improvements can be made and on the other hand forward connections are not the only way we can connect two oscillators. The following sections investigate possible improvements.

3.4.1 Momentum

Gradient descent it rarely used alone, a term of momentum is generally added. It has the good advantage to make convergence faster when error is large and to suppress or reduce potential oscillations of the weights during convergence. The momentum introduce some kind of weight's inertia by keeping a certain

percentage μ of the weight change at time $t-1$ when computing the change at time t. Equation 3.8 and 3.9 thus become:

$$\Delta a_{ij}^t = 2\,\eta\,\mathcal{E}_i \frac{x_j}{\left((x_j - \tilde{x}_j)^2 + v_j^2\right)^{1/2}} + \mu \Delta a_{ij}^{t-1} \tag{3.10}$$

$$\Delta b_{ij}^t = 2\,\eta\,\mathcal{E}_i \frac{v_j}{\left((x_j - \tilde{x}_j)^2 + v_j^2\right)^{1/2}} + \mu \Delta b_{ij}^{t-1} \tag{3.11}$$

3.4.2 Recurrent Links

Until now our oscillators were connected with one forward link only. However we can intuitively convince ourself that adding a link going in the opposite direction may reinforce the stability of the system and also make the convergence toward the right phase quicker. Indeed with recurrent links, oscillators i and j are both adjusted, instead of just i in the former case. Moreover we know that in coupled oscillators the amplitude change, but if j influence i and not the converse, then i will change its amplitude and j not, since i has no effect on j. If recurrent links are used, then hopefully both amplitudes will change in the same proportions.

The most straight forward way to add a recurrent link is represented in Figure 3.1(b). In this case two new weights are introduced and those may simply be updated according to Equations 3.10 and 3.11 with $\phi_{ji} = -\phi_{ij}$. However experiments show us that, instead of stabilising the system, this type of link rather introduces instability and even weights oscillations. The problem is that the path followed by one link in order to reduce the error is not symmetrical to the one followed by the other link, convergence is thus more difficult to achieve. Furthermore it has the additional drawback to introduce two new free parameters which in fact depend on those of the forward link. A better type of recurrent link is shown on Figure 3.1(c). In this case the parameters of the backward link depend on those of the forward one. Indeed since $\phi_{ji} = -\phi_{ij}$ there is an obvious dependency between the coupling parameters of the two links, in this case $a_{ji} = -a_{ij}$ and $b_{ji} = b_{ij}$.

3.4.3 Epoch Updating

When a coupling exists the system described by Equation 3.5 does not behave exactly like a sine and thus the error given by Equation 3.7 is not strictly constant but rather oscillate. Indeed if we superimpose a sine wave on the oscillator's trajectory, some points will match up and some not. Minimising the error at each time step is thus not enough, because at another time step the error may have increased. Fortunately the oscillator's trajectory is most of the time very close to a sine and our algorithm thus does its job correctly.

However in transient situations were, for example, the system is perturbed by an external force or during gaits change, the algorithm may be fooled because the behaviour of the nonlinear equations may be far from a sine. The momentum may then impose great changes to the coupling parameters, thing which is not advisable in those situations. In fact changing a_{ij} and b_{ij} too frequently is not an advisable thing to do at all. Here is where epoch updating plays its role,

	ϕ_{12}	ϕ_{43}	ϕ_{14}
walk	0.75	0.75	0.5
trot	0.5	0.5	0.5
bound	0.5	0.5	0.0

(a)

	walk	trot	bound
walk	–	2.2	4.6
trot	1.9	–	3.4
bound	3.4	2.9	–

(b)

Table 3.1: (a) Three types of gait in quadrupeds and their associated phases differences. (a) Mean time (in seconds) needed to change form the *row* gait to the *column* gait.

instead of correcting the coupling parameters at each time step, the error is accumulated and changes occur after a given elapsed time. This time does not need to be very large, one tenth of an oscillator's period is enough, but it should obviously not goes beyond one period.

3.5 A Toy Example

In order to illustrate the power of our adaptive algorithm, we will study here its behaviour on a toy example, quadruped locomotion. Three types of locomotion are commonly observed in quadrupeds, namely walk, trot and bound. Those gaits result from a precise timing between the four legs i.e. precise phases differences must be maintained between the four oscillators controlling the motion of the legs [5]. If we take the convention to number the oscillators in the following order: left front, left hind, right hind and right front, starting from 1, then the required phase differences needed to generate the three gaits are given in Table 3.1(a).

For this test the frequency of the legs is set to 1 Hz, their amplitude to 15 degrees, $\alpha = 2$ and $\tilde{x}_i = 0$ for all i. The adaptive algorithm uses momentum with $\eta = 10^{-3}$, $\mu = 0.7$ and epoch updating occurs after 0.1 s i.e. one tenth of a period. Moreover oscillators are coupled in the following way (σ_1, σ_2), (σ_4, σ_3), (σ_1, σ_4) and (σ_2, σ_3), with the type of recurrent links shown on Figure 3.1(c). Table 3.1(b) shows the average time required by our adaptive algorithm to change from one gaits to another. This is no surprise that the transition from walk to bound and conversely is the most difficult, then comes trot-bound and finally walk-trot. Please note that those values are rather indicative since the time needed to switch the gaits does not seem independent of the moment at which the switch occurs. Additionally Figure 3.2 shows, for each possible transition, how the phases are modified. Figures 3.3, 3.4 and 3.5 show the evolution of the coupling parameters when a switch of gait occurs. It is interesting to note that different values of the coupling parameters may produce the same phase difference, this is particularly visible on Figures 3.4 and 3.5 where the walk gait is involved. In fact whereas the phase difference stays the same, the amplitude change. Indeed if we look at equation 3.7 we see that no constraint is imposed on the amplitude A_i. In order to prevent too large variations of the amplitude we could have set $A_i = \beta\sqrt{E_i}$ in Equation 3.6 and $\beta \approx 1$[1], however experiments

[1] If $\beta = 1$ then the terms related to the phase difference ϕ_{ij} in Equation 3.7 vanish.

show that this approach is less efficient. Moreover, finding a way to somehow introduce amplitude limitations in Equation 3.7 is very important in order to ensure the convergence. Indeed if we look at Figure 3.4 we may easily see that, after some point, some a_{ij} tend to grow linearly with time, things that should be avoided since when those experiments were carried out, the amplitude had already increased of 5 degrees.

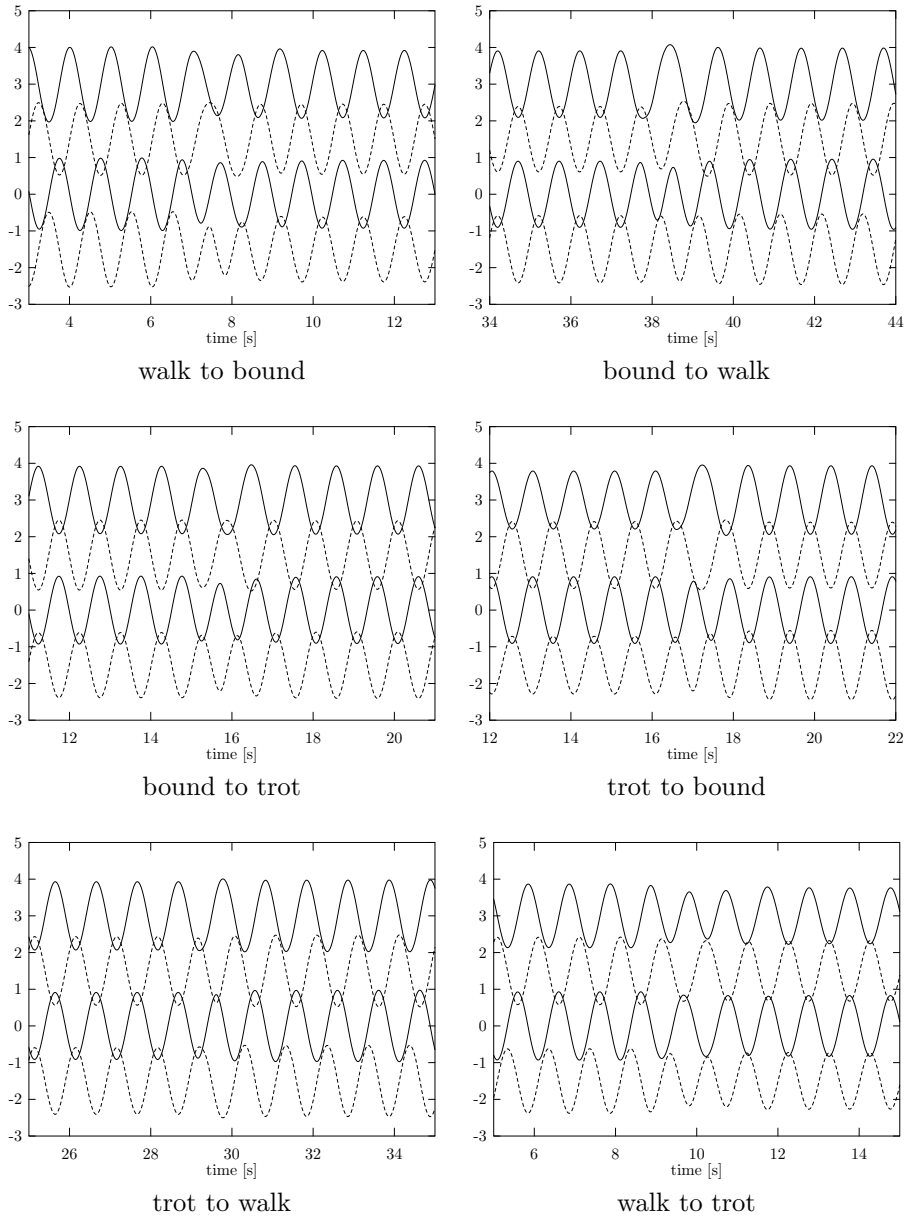


Figure 3.2: Activity of the four oscillators for all possible gait transitions. Curves correspond to (from top to bottom): LF, LH, RH and RF oscillators respectively.

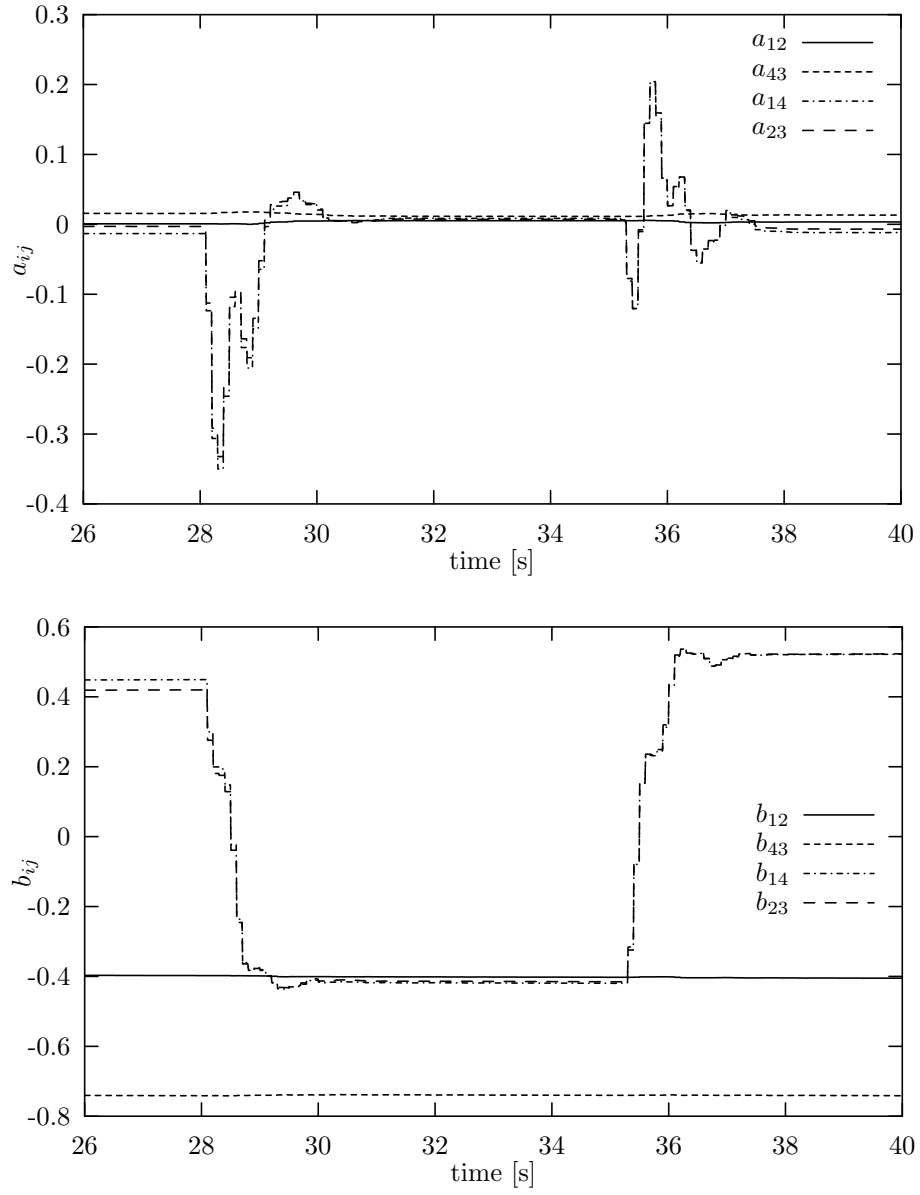


Figure 3.3: Evolution of the coupling parameters a_{ij} and b_{ij} when the gait is switched back and forth from bound to trot and back.

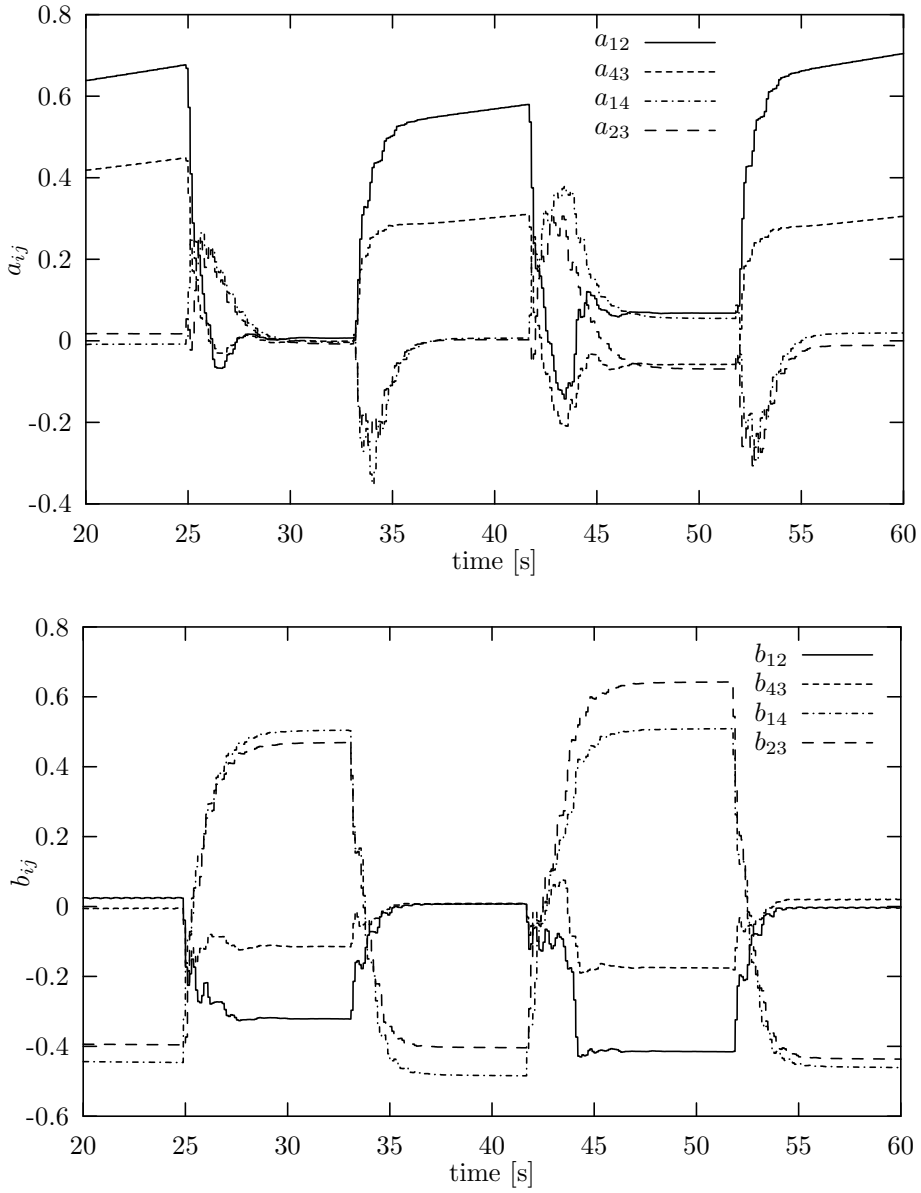


Figure 3.4: Evolution of the coupling parameters a_{ij} and b_{ij} when the gait is switched back and forth from walk to bound and back.

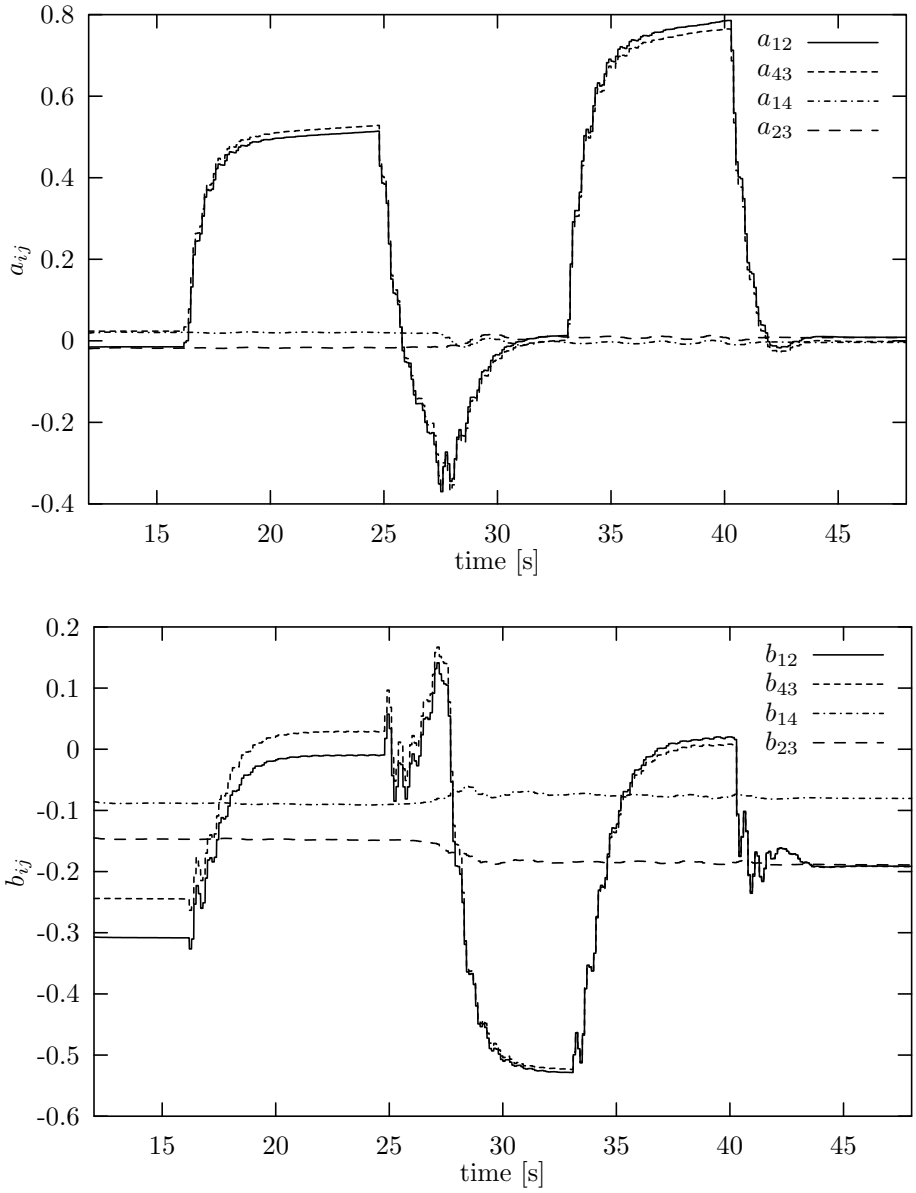


Figure 3.5: Evolution of the coupling parameters a_{ij} and b_{ij} when the gait is switched back and forth from trot to walk and back.

Chapter 4

Self-organisation of Locomotion

At this point we possess a group of randomly interconnected modules and an adaptive algorithm which allows us to specify precise phase differences between two coupled oscillators. What we do not know however is the amplitude and reference position ($\tilde{x}$) of each oscillators as well as how to connect them and what phase differences are required to achieve good locomotion. For a structure like the one shown on Figure 4.1 where each element possesses two oscillators and each one is connected to its nearest neighbours, there are 176 free parameters. Of course a nearest neighbours wiring scheme is certainly not optimal, for example our quadruped bug (Section 3.5) required that the four legs were connected together.

4.1 Genetic Algorithms

From the ground breaking works of Karl Sims back in 1994 [23, 24] to the more recent ones of Jordan Pollack [18], good locomotion behaviours have been obtained with the help of a genetic algorithm or designed by hand (see Section 1.1). In our case, using a genetic algorithm has the drawback that the overall process of self-organisation is split in two parts, on one hand modules move freely and connect themselves in a random way and on the other hand a group is by some mean selected and evolved. However the evolved parameters are highly specific to the structure of the group and thus fails to give a satisfactory solution to our problem. Genetic algorithms could however give us a clue to how locomotion can be achieved in a particular group when nearest neighbours coupling is used, it should at least shows us if locomotion is possible.

4.1.1 A Simple GA

The most straight forward way to implement a GA which optimises the total distance covered by a group of modules during a defined time is to build a

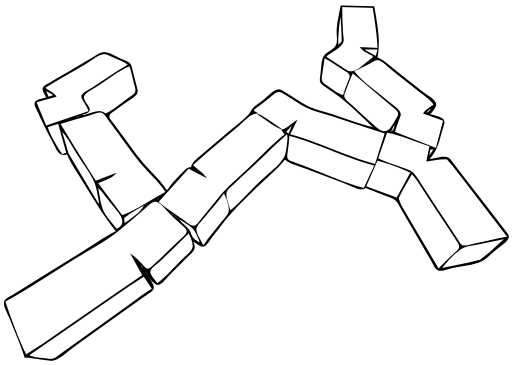


Figure 4.1: A group of eight interconnected modules obtained by brownian motion.

population of genomes, each composed of 176 floating points values and let it evolve during a sufficiently high number of generation. The genome is simply the repetition of sets of the form $\{E_i, \tilde{x}_i, C_i\}$ for all oscillators i, where C_i is the set of coupling parameters and is defined by $\{(a_{ij}, b_{ij}) | j \in \mathcal{N}_i\}$, with $\mathcal{N}_i$ containing the index of the nearest neighbours of i. For the group of modules shown on Figure 4.1, we have six modules which possess two neighbours and two possessing only one. Since there are two oscillators per modules we have a total of sixteen oscillators. One neighbour implies four connections and since there are two parameters (a_{ij} and b_{ij}) per connections, we thus have a total of $(6 \cdot 2 \cdot 4 \cdot 2 + 2 \cdot 4 \cdot 2 = 112$ parameters used for inter-modules coupling and $8 \cdot 2 \cdot 2 = 32$ parameters used for intra-modules coupling—the two oscillators in a module should also be connected. We finally add two additional values per oscillator (E_i and $\tilde{x}_i$) which raise the total to $112 + 32 + 16 \cdot 2 = 176$ floating point numbers which are, for practical reasons, normalised between 0 and 1.

The crossover operator takes two parents and for each position exchange the genes at that position with probability 1/2. Mutation randomly selects a position and changes the corresponding value. The new value are selected according to an uniform distribution. The GA is a *Steady State* GA where 70% of the population is replaced at each generation by children of the remaining 30% if and only if they get a better fitness i.e. for a population of 100 genomes, at each generation 70 children are generated and they replace the worst genomes if their fitness is better.

The evaluation function is based on the distance covered by the group's centre of gravity during the time of the evaluation. Each genomes is evaluated during 30 cycles with the oscillators working at a frequency of 1 Hz. Moreover when the genome is decoded, each normalised value is transposed into the following range of validity, $\tilde{x}_i \in [-\pi/4, \pi/4]$, a_{ij} and $b_{ij} \in [-2, 2]$ and $|\tilde{x}_i| < \sqrt{E_i} + |\tilde{x}_i| \leqslant \pi/2$. This last constraint prevents oscillators to go beyond

the limits set on the joint motors. Indeed although a small bounciness factor is set on the stops, if the angular velocity with which the motor reaches the stops is high, large forces may be introduced into the system and thus produce instabilities.

The first runs of the GA with a population of 100 individual and a crossover and mutation probability of 0.9 and 0.05 respectively, gave more problems than solutions. First of all some genomes tend to make the physical simulation explode. When such kind of problems arise in physical simulation it is most often due to error in numerical integration i.e. forces are not correctly computed anymore and the robot starts to jump erratically and finally explodes. To cope with this problem a small integration step is generally required, a value of 0.001 seconds proved to be a good compromise between speed and accuracy, but it however does not totally suppresses explosions, we thus needed a way to detect them. Fortunately it appears that in ODE the position of an object after an explosion is $(\text{Nan}, \text{Nan}, \text{Nan})$[1], thus a call to the GNU[2] C function `isfinite()` allows us to easily detect ill–behaving robots and a null fitness was then attributed to those individuals.

The second problem encountered was the fact that some individuals take a very long time to be evaluated. This proved to be caused by some combination of coupling parameters a_{ij} and b_{ij} that generate divergent or saturating oscillators' behaviours. Indeed singularities in the integration of the nonlinear differential equations are then introduced and the integrator—a 4th order Runge-Kutta-Fehlberg method with 5th order error estimate—takes more time to correctly compute the following values. Since of course diverging solutions hardly produce good locomotion behaviours they could be eliminated in the same way as exploding solutions, however before being eliminated diverging systems must be detected. A first solution was to test the whole oscillatory network during 10 cycles and then attribute a null fitness to diverging systems. Unfortunately most of the genomes of the first generations had at least one diverging oscillator and thus got a null fitness. This problem is however far less consequent than the one that arise after running the system during some generations.

The best individual obtained after about hundred generations was able to cover more than 5.5 meters in 30 seconds which is equivalent to an average speed of 0.18 mps. However this performance is achieved with the help of numerical integration errors in the simulation. Indeed since the system works in open-loop i.e. no feedback from the world, oscillators continue to force the joint motors to a given position, even if this one cannot be reached due to friction with the ground or with other modules. Numerical integration then introduces new internal forces that tend to grow just enough to make the robot jump and bounce around, but not explode. The distance finally covered by the robot is then sufficiently large to defeat any well behaving solution.

4.1.2 Another GA

The space explored by the previous GA was huge. In fact it can be easily reduced by taking into account that locomotion implies synchronisation and

[1] Nan means not a number.

[2] GNU is here important since the function `isfinite()` does not exist on Solaris for example.

that we possess an adaptive algorithm which is, at least theoretically, able to synchronise two oscillators at a given phase difference. Instead of defining a_{ij} and b_{ij} directly in the genome, those parameters may be replaced by the desired phase differences that the adaptive algorithm should reach. Doing so reduce the size of genome to 68 instead of 176. Actually one may think the size should be $144/2+32 = 104$, however since we know that $\phi_{ji} = -\phi_{ij}$, the phase differences must be defined in only one direction and since there is $n-1$ non-oriented inter-modules connections in a tree–shaped group of n modules, we get a total of $7 \cdot 4 + 8 = 36$ phase differences—$7 \cdot 4$ is the number of inter-modules phase differences and 8 is the number of intra-modules ones. The remaining 32 values come from the internal parameters $\tilde{x}_i$ and E_i of each oscillators.

Although the behaviours obtained are much less erratic than in the previous GA, in many cases the adaptive algorithm is however not able to correctly reduce the error and even sometimes it is increased. This problem seems to arise when two too different constraints are set on an oscillators. Indeed let's think of two oscillators, both connected to a third one, then let's suppose that the first has a phase $\phi_1 = 0$, the second $\phi_2 = \pi/2$ and that $\phi_{31} = 0$ and $\phi_{32} = 0$, thus $\phi_3 = \phi_1 + \phi_{31} = 0$ and $\phi_3 = \phi_2 + \phi_{32} = \pi/2$ which is impossible to satisfy. Hopefully a recurrent connection between the oscillators 1 and 2 will change their phase in order to have $\phi_{12} = 0$, this is in fact what appends is our carefully designed quadruped bug (Section 3.5), but with random phase differences things are rarely so easy. A way toward the resolution of conflicting constraints may be to take care that each module synchronises itself with one of its neighbour only. However although this may solve conflicting constraints in a tree–shaped robot, it will not work when loops exists, because in this case the sum of phases differences around the loop should be zero. Anyway if the adaptive algorithm is used, an important point still remains, self-organisation will occur at two levels at the same i.e. the level of the adaptive algorithm and the level of the GA, we thus need to be sure that no problem arises from the adaptive algorithm since otherwise the GA may not correctly evaluate the individuals. We must keep in mind that the GA is just used to somehow explore the space of all possible parameters, thus after looking at the numerous difficulties it introduces and since a GA is essentially a global optimisation procedure which is very often used for this kind of task, we should rather try to find new, local algorithms which are able to find good locomotion behaviours. It should be noted here that the development, test and execution of a sufficiently robust GA took at least one month, but gave no satisfactory results at all.

4.2 Random Search

The main drawback of genetic algorithms is that they optimise a given group of modules and do not provide any clue at all on how the locomotion of a randomly assembled structure can be achieved. Since the main novelty of our modular robots is their ability to move by themselves and to form random structures that try to optimise their own motion, using a global algorithm like a GA on a somehow selected group of modules is far from satisfactory. Moreover although the search space may be reduced by replacing coupling parameters by phase differences, our GAs still need to use a fixed connection schemes, because

adding where each oscillators should be connected into the genome will make the problem intractable. How oscillators are connected is a quite important point, we thus do not want to limit ourselves to only nearest neighbours connection scheme. Moreover since we are potentially able, thanks the adaptive algorithm, to find the coupling parameters for a given phase difference, we can temporally replace our nonlinear oscillators by simple harmonic ones of the form:

$$x_i = A_i \sin(2\pi\nu t + \phi_i) + \tilde{x}_i \tag{4.1}$$

where ν is the same for all oscillators and the free parameters are A_i, ϕ_i and $\tilde{x}_i$. Of course what an optimisation algorithm should do is to find the best values for the free parameters of each oscillators, but using harmonic oscillators has the great advantage of moving coupling parameters or phase differences out of the way. Of course oscillators are no more really connected, but phase differences may however always be found since $\phi_{ij} = \phi_i - \phi_j$. A simple local algorithm may thus be defined by repeating the following procedure:

1. Each module randomly selects three parameters for both oscillators according to the current best set of parameters,

2. During some time, each module change smoothly from its current parameters to the newly selected ones,

3. The current position $\mathbf{p}_0$ of the group is recorded and the evaluation begins,

4. After some given time the current position $\mathbf{p}_1$ of the group is computed and compared to the starting one. The fitness is then defined as $||\mathbf{p}_1 - \mathbf{p}_0||$.

In the first step, the ranges of validity of each parameter are the same as those given in Section 4.1.1. The second step is mandatory because Equation 4.1 contrary to Equation 3.5 does not give a smooth behaviour when the set of parameters is changed. Indeed if changes occur too quickly, the joint motors are forced to reach the new position instantaneously, thing which is not possible. Instabilities are then introduced in the physical simulation and robots start jumping around or explode. Parameters thus need to be slightly modified until they reach the desired value. The two remaining steps constitute the evaluation procedure, since good locomotion capabilities means going as far as possible, the fitness of a given set of parameters is proportional to the distance covered by the group during a certain number of cycle.

The random selection of new parameters is obviously the crucial point of the algorithm. What is the best way to generate new parameters? For example new values may be selected according to a gaussian distribution centred on the current best solution, and if such a change proved to be useful, the distribution may be afterwards somehow biased in the direction of this change. A great number of appealing procedures may be used to produce a random walk in the search space, but in our case one of them deserves extensive interest and is described in the following section.

4.3 Simulated Annealing

Simulated annealing is an optimisation algorithm proposed by Kirkpatrick et al. back in 1983 as a way to optimise the design of computers (wiring, placement of components,...) [15]. It takes its roots in combinatorial optimisation and statistical mechanics and particularly on how crystal are obtained from a melt. Indeed in a crystal, atoms are at their lowest possible energy level and this state is generally obtained through careful annealing (a slow reduction of the temperature) methods. Similarly we may see a set of parameters as being the atoms of a melt, the energy as the function to optimise and the annealing as the optimisation procedure. At the beginning of the process the temperature is sufficiently high to allow the system to jump at energy levels far away from the initial one, then the temperature is slowly decreased and the possible energy levels get more limited, finally if the temperature is sufficiently low and the annealing was correctly done, then system should rest at a level of minimal energy.

Since the early work of Kirkpatrick, simulated annealing as been greatly enhanced and gave birth to three new algorithms, namely Boltzmann, fast and, more recently, very vast annealing (or adaptive simulated annealing) [14]. From a formal point of view the main benefit of those algorithms is that their convergence is assured if a certain temperature schedule is followed, it could thus be very useful for finding the optimal locomotion behaviour of a group of modules. Simulated annealing is based on the interplay of the following functions:

1. $g(\Delta x)$ gives the probability distribution of the distance between the set of newly generated parameters ($\mathbf{x}_1$) and the current one ($\mathbf{x}_0$),
2. $E(\mathbf{x})$ gives the energy associated with the set of parameters $\mathbf{x}$,
3. $h(\Delta E)$ gives the probability distribution of the difference between $E(\mathbf{x}_1)$ and $E(\mathbf{x}_0)$,
4. $T(k)$ gives the temperature at the annealing iteration k.

The simplest implementation of simulated annealing uses the following functions for $h(\Delta E)$ and $T(k)$:

$$h(\Delta E) = e^{-\Delta E/T_k}$$

$$T(k) = \frac{T_0}{k\,\mu_T}$$

and $g(\Delta x)$ is generally a uniform or gaussian distribution which may vary with the temperature or not. Since the development of faster algorithms, classic simulated annealing is not used anymore, we will thus turn ourselves to Bolzmann annealing instead. This one use the following functions:

$$g(\Delta x) = (2\pi T)^{-D/2}\, e^{-\Delta x^2/(2T)}$$

$$h(\Delta E) = \frac{1}{1 + e^{\Delta E/T}}$$

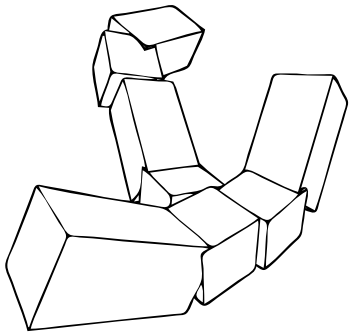


Figure 4.2: Small group of four interconnected robots. This structure is actually obtained by cutting the group shown on Figure 4.1 in two parts.

$$T(k) = \frac{T_0}{\ln(k)}$$

where D is the size of the parameters set. It has been shown that as long as the given temperature scheduling is respected, a global optimum is always found [14].

4.4 Results

Since the overall procedure followed by simulated annealing in order to optimise a given energy function $E(\mathbf{x})$ is well known, it will not be described here[3] Moreover since in our case we are looking for the set of parameters that make a group of modules cover the longest possible distance, we will maximise the energy function instead of minimising it. Actually this can also have been done by minimising $E(\mathbf{x}) = 1/(1 + \delta(\mathbf{x}))$, where $\delta(\mathbf{x})$ is the distance covered by the robot during a given time step.

Although Boltzmann annealing is faster than classical annealing it still takes a rather long time to find the optimal solution. To cope with this problem we will take the small group shown on Figure 4.2 instead of the big one that was used with the GAs (Figure 4.1). Figure 4.3 shows the results of at typical run that begins at temperature 0.022 and evaluates 25 sets of parameters before incrementing k. The best solution obtained so far was covering a distance of 2.46 meters in 20 seconds which means an average speed of 0.123 mps. However if we look at the solution we see that, once again, it takes advantage of numerical instabilities to jump far away from its initial position. Fortunately very good solutions were obtained, those cover a distance of slightly less than one meter in

[3]For example, the *GNU Scientific Library* (GSL) provides an implementation of a basic simulated annealing. The corresponding code can be found in the file siman.c. It is then easy to change $h(\Delta E)$ and the temperature scheduling to implement Boltzmann annealing.

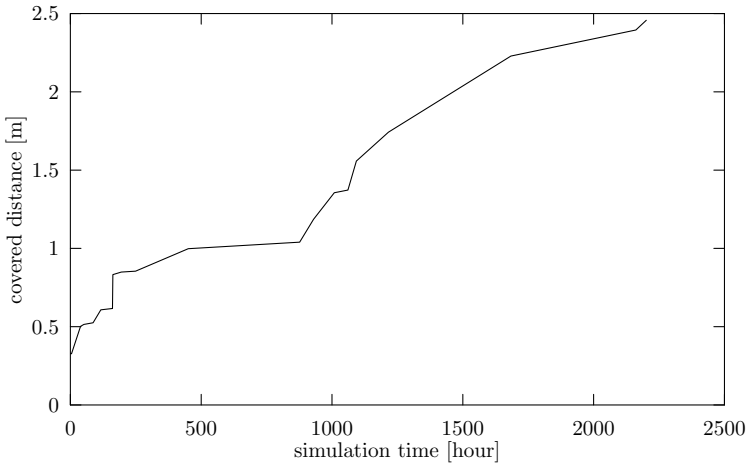


Figure 4.3: Evolution of the distance covered by the best solution in function of the simulation time.

20 seconds, but do it in smart way[4]. Although solutions are appealing, the big point is the time taken to find them. Indeed we see on Figure 4.3 that about 500 hours (more than 20 days) are required to find solution covering about one meter. Of course simulation of such a simple robot goes very quickly and in about one hour of real time, 500 hours have already been simulated, however if simulated annealing was used as this in a real time robots it would surely be boring and useless.

4.5 Last Remarks

Both genetic algorithms produced more problems than solutions, but those problems however allowed a number of bugs and misconceptions to be discovered in the code. It could surely have worked better by adding complexer functions of fitness or by checking more extensively to prevent bad behaviours or even setting a certain parameter differently, but after nearly one month of patch and corrections it was still not working very well. For just an exploratory process it took a rather long time.

Simulated annealing however produces better results even if implementing the algorithm, and particularly the evaluation function, was not straight forward. Indeed, as was previously explained in Section 4.2, if the transition is not slow enough, instabilities arise in the simulations. However the overall benefit of using harmonic oscillators highly compensate implementation difficulties. In fact the GAs would certainly have worked better if the same type of oscillators where used, even if it appears very lately (two weeks before the end of the project) that a misuse of the dBodyDisable function in ODE may be at the root of an undefined number[5] of instabilities. As previously mentioned mod-

[4]See the web page of the project at `birg.epfl.ch` for videos.
[5]Undefined because there was no more time to go back to GAs.

ules' sticks-wheels are disabled at this step, because they are no more useful and simulating them is quite complex and thus takes time. The ODE function dBodyDisable is thus called on the wheels at the beginning of the simulation, but the tricky point is that a disabled body which is attached to an enabled one becomes enabled at the next time step. This is the reason why until recently all simulations done with the group of modules shown on Figure 4.1 were actually simulating the wheels even though they were not drawn on the screen. Of course this does not explain all instabilities since after that bug was corrected some still happen.

Chapter 5

C++ Framework

Modular robotics is a vast field of research and since this project just deal with a small percentage of the problems that may arise, the code developed was made as reusable as possible. This chapter documents the main classes and methods which were developed to facilitate the coding of programs dealing with the modules described in this document.

5.1 The **Module** Class

Constructor

Module(dWorld &w, dHashSpace &s, bool legs=true)
Create a new module which is linked to ODE world w and ODE space s. If legs = true, the sticks-wheels are simulated.

Utility Methods

void **draw**(void)
Draw the module on the screen.

void **move**(dReal dx, dReal dy, dReal dz)
Move the module by the vector (dx, dy, dz) relatively to its current position.

void **rotate**(dReal angle, int axis=3)
Rotate the module around axis. Axis 1 is x, 2 is y and 3 is z.

Set Methods

void **setAngle**(int axis, dReal angle)
Set the angular position of one of the two joint motors, axis=1 or 2.

void **setSpeed**(int leg, dReal speed)
Set the angular velocity of one of the two module's legs, leg=1 or 2.

void **setAttach**(bool f)

If f=true then the module may attach itself to other ones.

void **setMode**(int m)
Set the current activity mode of the module. Mode 0 is random walk, mode 1 is used when switching from mode 0 to mode 2 and mode 2 is external control (oscillators for example). Once in mode 2, joint motors as well as wheels can be controlled through the functions setAngle and setSpeed respectively.

Get Methods

dReal **getMass**(void)
Return the module's total mass.

void **getCMass**(dVector3 c, bool pos=true)
If pos=true the position of the module's centre of mass is returned. If pos=false the velocity of the module's centre of mass is returned.

dBodyID **getBodyByFace**(int f)
Return the body associated with face f.

dBodyID **getBodyID**(int index)
If index=0 (1) this function returns the dBodyID of the long (small) part of the calling module.

int **numOfNeigh**(void)
Return the number of modules connected to the calling one.

int **getId**(void)
Return the ID of the calling module.

Main Methods

void **next**(void)
This function is particularly useful when module operates in mode 0, because it automatically control the wheels in order to generate a random walk. It automatically change the mode when a module get connected.

bool **isValid**(dBodyID bdy_id1, dBodyID bdy_id2, int &f1, int &f2)
This function is called after a contact occurred between bdy_id1 and bdy_id2. It returns true if docking occurs and false otherwise. In case of success the two faces to be linked are returned in f1 and f2.

void **link**(dBodyID bdy_id1, dBodyID bdy_id2, int f1, int f2)
Create a joint between the body bdy_id1 of the calling module and the body bdy_id2 of another module. Where faces f1 and f2 were determined by **isValid**.

void **neigh**(list<Module*> &l)
Fill the list l with pointers on the modules connected to the calling one.

Save and Load Methods

void **save**(ostream &out)
Save the position, orientation and neighbours' id of the calling module through the stream out.

void **load**(istream &in)
Load the data recorded with save from the stream in.

Class Methods

void **setAvSpeed**(float s)
Set the average speed of the wheels to s mps.

void **setDMax**(float d)
Set $d_{\max}$, see Section 2.2.2. Default is 0.24 cm.

void **setLMax**(float l)
Set $l_{\max}$, see Section 2.2.2. Default is 0.4 cm.

void **setTMax**(float t)
Set $\delta_{\max}$, see Section 2.2.2. Default is 7 degrees.

5.2 The Osc Class

Constructor

Osc(void)
Create a new oscillator.

Set Methods

See Equation 3.5 for the meaning of the variables.

void **setPos**(double p)
Set x_i.

void **setVel**(double v)
Set v_i.

void **setStPos**(double stp)
Set $\tilde{x}_i$.

void **setEnergy**(double e)
Set E_i.

void **setFreq**(double f)
Set the frequency $\nu = \frac{1}{2\pi\tau}$.

void **setTau**(double t)
Set τ.

Get Methods

double **getPos**(void)
Return x_i.

double **getVel**(void)
Return v_i.

double **getStPos**(void)
Return $\tilde{x}_i$.

double **getEnergy**(void)
Return E_i.

double **getFreq**(void)
Return the frequency $\nu = \frac{1}{2\pi\tau}$

double **getTau**(void)
Return τ.

double **getAcc**(void)
Return $\ddot{x}_i$.

double **getAmpl**(void)
Return the current estimated amplitude $A_i = \left((x_i - \tilde{x}_i)^2 + v_i^2\right)^{1/2}$.

double **getBeta**(void)
Return the current relative amplitude grow $\beta_i = \frac{A_i^2 - E_i}{E_i}$.

double **getPhase**(void)
Return the current estimated phase ϕ_i. This value is computed by assuming that $x_i = A_i \sin(2\pi\nu t + \phi_i) + \tilde{x}_i$.

double **wSum**(void)
Return $\sum_j \Gamma_j$.

Main Methods

void **nextState**(double t, double dt)
Integrate the system between t and t+dt, but does not update the position and velocity i.e. a call to getPos or getVel returns the same value as before. This is needed since the state of all oscillators should be updated synchronously.

void **update**(void)
Set the current position and velocity to the values computed by nextState.

Class Methods

void **setAlpha**(double a)
Set α.

double **getAlpha**(void)
Return α.

5.3 The **Link** Class

Constructors

Link(int size = 1)

Create a new link that possesses only one weight. Link is a generic class that link two Node, thus it has no way to now how many weights are needed. If a link connects two Osc then size must be equal to 2.

Link(Node *in, Node *out, int size=1)
Same as the previous constructor, but additionally connects the two nodes together.

Set and Get Methods

void **setWeight**(double w, int index = 0)
Set the value of the weight at index.

double **getWeight**(int index = 0)
Return the value of the weight at index.

Main Methods

void **connect**(Node *in, Node *out)
Connect two nodes together.

double **wInput**(int index = 0)
Return the weighted input for the corresponding index. It call getValue with the same index on the input node and multiply the results by the corresponding weight.

5.4 The AdaptLink Class

Constructors

Constructors for this class are the same as those of the Link class, however they take Osc instead of Node as parameters.

Set Methods

void **setEpoch**(double t, double dt)
Set the last epoch time and the elapsed time between epoch to t and dt respectively.

void **setEta**(double e)
Set the learning constant η.

void **setMomentum**(double m)
Set the momentum constant μ.

Main Method

void **update**(double t, double dt, double p, bool epoch=false)
Update the weights according to Equations 3.8 and 3.9. If epoch = true, then epoch updating is used.

5.5 The **World** Class

Constructor

World(void)
Construct a new world. This class encapsulates everything that is needed to simulate modules and their interactions, namely docking, collision detection and friction.

Set and Get Methods

void **setGravity**(dReal g)
Set the gravity along the z axis.

void **getModCMass**(dVector3 c, bool pos = true)
If pos = true (false), then return the position (velocity) of the center of mass of the whole group of modules.

Main Methods

void **addModule**(Module *m)
Add a new module to the world.

void **next**(double t, double dt)
Call nextStep with the same parameters on each module.

void **draw**(void)
Draw the modules and the ground.

void **apply**(void (*f)(Module*, void*), void *data = 0)
Apply the function f on each module. When f is called, the first argument is a pointer on the module and the second is data.

void **centerMod**(void)
Move all modules in order to have the center of mass of the whole system at $(0, 0, 0)$.

Save and Load Methods

void **save**(ostream &out)
Save informations about the world into the stream out.

void **load**(istream &in, bool withLegs=true)
Read the previously recorded data from the stream in. withLegs = true means that information concerning the legs are contained in the data. Do not set withLegs to true if the saved modules did not use legs.

Chapter 6

Conclusion and Future Work

This chapter describes some of the tools used for this project. It then sums up what as been done and what remains to be done and finally draws a broad conclusion.

6.1 Used Tools

Everything from the start to the end of the project was done with open source softwares on an open source operating system, namely Linux. The following list sums up the main tools that were used and where they can be found:

- As previously mentioned, physical simulation were exclusively done with the *Open Dynamics Engine version* 0.039. Although this software is still in development it proved to be very accurate and stable. `http://opende.sourceforge.net`
- Nonlinear equations integration as well as random numbers generation were done with the *GNU Scientific Library* (GSL). GSL is a huge library that implements most of the common mathematical things that someone would use in its programs. For example, it proposes ten types of ordinary differential equations integrator and can generate random numbers from the most common distributions. `http://www.gnu.org/software/gsl`
- Genetic Algorithm were done with *GAlib*, an object oriented and highly parameterisable genetic algorithm library. GAlib implements the most common GAs, types of genome, crossover and mutation functions. If nothing suit your needs you can always program it yourself since GAlib is highly extensible. `http://lancet.mit.edu/ga/`
- Since GAs work better with a large population, two weeks were consacred to the development and test of a parallel genetic algorithm. Since the machines on which the distributed GA was running were heterogeneous

and not exclusively dedicated to this task, the parallelisation part was done with the *Parallel Virtual Machine* (PVM) version 3. With PVM, new machines can be easily added or removed during computation and hosts failure are automatically detected and reported. `http://www.netlib.org/pvm3`

- This document was written with LaTeX 2$_\varepsilon$ and plots were generated by *GNUplot* with MetaPost output. Drawings were all generated by the mix of two images resulting from a Sobel edge detection algorithm applied one the normal and depth map of the corresponding 3D rendering. The final image was then vectorised with the *autotrace* program. `http://autotrace.sourceforge.net`

6.2 Future Work

Although, during the four months that last this project, a wide variety of questions have been addressed, a huge work still remains to be done. First the adaptive algorithm need to be more extensively tested in order to verify its stability and convergence properties. For example we may conduct a number of tests where coupling, oscillators position and velocity are randomly generated and then tested during at certain number of cycles. Those kind of experiments would however require a way to solve the problem of the loops arising in a randomly connected graph. Once a well behaving algorithm is found the problem of setting the coupling parameters a_{ij} and b_{ij} will then be resolved, but that point will however still require to know how to connect the oscillators. Indeed we got round this problem by replacing nonlinear oscillators by harmonic ones, we thus also replaced relative phase differences by global ones. This change allowed us to find, with help of simulated annealing, that interesting solutions may be obtained rather quickly. However those solutions still work in open-loop i.e. no feedback from the environment exists. This is not desirable in real world applications, the ultimate step is thus to find an efficient algorithm that, while continuously integrating sensors inputs, is able to optimise not only the individual parameters of each oscillators, but also the coupling between them. We could then bring all those algorithms together and finally build what was somehow the main goal of this project, a self-building and self-organising modular robots that optimise their locomotion.

6.3 Conclusion

Although the field of modular robotics is quite appealing, it has proved to be a difficult one. If we look at results presented in this document we easily see that nothing work in all situations, for example the adaptive algorithm does not converge like it should or, for some individuals or sets of parameters, the GAs and simulated annealing are completely fooled by numerical artifacts. In the first chapter are put forward some advantages of using simulation instead of real world implementations, but even if those good points obviously remain, at the time of this conclusion some drawbacks need to be added. The main one

is probably the one that is associated with physical simulations. Indeed since this project starts from the ground, everything has to be build and thoroughly tested, however even after the smallest modification, new kind of problems arise and had to be fixed. For example, in order to be able to run a GA which optimise a not so complex group of robot like the one shown on Figure 4.1, it takes about two weeks. However, although the GA never work sufficiently well, it allowed numerous bugs and misconceptions to be found and corrected. Furthermore the late results obtained with simulated annealing tend to show that the code seems finally stable.

From a more optimistic point of view, this project somehow set out the ground for future researches in the field of modular robotics. It has explored how random robots may be build from the interaction of numerous identical modules, it also showed how, through a simple adaptive algorithm, groups of oscillators may be quickly synchronised, and finally proposed a way to automatically optimise the locomotion of a modular robot. Moreover, thanks to the development of a C++ framework, further experiments with modular robots should now become easier and could hopefully encourage somebody to continue the project where I leave it today.

References

[1] J. Buchli and A.J. Ijspeert. Distributed central pattern generator model for robotics application based on phase sensitivity analysis. In *Proceedings of the First International Workshop on Biologically Inspired Approaches to Advanced Information Technology – Bio-ADIT2004*, Lecture Notes in Computer Science. Springer, January 2004.

[2] Z. Butler, R. Fitch, and D. Rus. Distributed goal recognition algorithms for modular robots. In *Proceeding of the 2002 IEEE International Conference on Robotics and Automation (ICRA'02)*, 2002.

[3] Z. Butler, S. Murata, and D. Rus. Distributed replication algorithms for self-reconfiguring modular robots. In *Proceedings of the 6th International Symposium on Distributed Autonomous Systems (DARS'02)*.

[4] A. Castaño and P. Will. Representing and discovering the configuration of conro robots. In *Proc. of Intl. Conf. on Robotics and Automation*, May 2001.

[5] J.J. Collins and S.A. Richmond. Hard-wired central pattern generators for quadrupedal locomotion. *Biological Cybernetics*, 71(5):375–385, 1994.

[6] D. G. Duff, M. Yim, and K. Roufas. Evolution of polybot: A modular reconfigurable robot. In *Proc. of the Harmonic Drive Intl. Symposium*, Nagano, Japan, November 2001.

[7] T. Fukuda and Y. Kawauchi. Cellular robotic system (cebot) as one realization of a self-organizing intelligent universal manipulator. In *Proceedings of the 1990 IEEE Conference on Robotics and Automation*, pages 662–667, Cincinnati, May 13-18 1990. IEEE Computer Society Press.

[8] A. T. Hayes, A. Martinoli, and R. M. Goodman. Swarm robotic odor localization: Off-line optimization and validation with real robots. In D. McFarland, editor, *Special issue on Biological Robotics*, volume 21 of *Robotica*, pages 427–441. Cambridge University Press, 2003.

[9] A. Howard, M.J. Matarić, and G.S. Sukhatme. An incremental deployment algorithm for mobile robot teams. In *IEEE/RSJ International Conference on Robotics and Intelligent Systems*, pages 2849–2854, Lausanne, Switzerland, Oct 2002.

[10] A. Howard, M.J. Matarić, and G.S. Sukhatme. An incremental self-deployment algorithm for mobile sensor networks. *Autonomous Robots Special Issue on Intelligent Embedded Systems*, 13(2):113–126, 2002.

[11] A. Howard, M.J. Matarić, and G.S. Sukhatme. Network deployment using potential fields: A distributed, scalable solution to the area coverage problem. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems*, Jun 2002.

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

[13] A.J. Ijspeert, A. Martinoli, A. Billard, and L. M. Gambardella. Collaboration through the exploitation of local interactions in autonomous collective robotics: The stick pulling experiment. *Autonomous Robots*, 11(2):149–171, 2001.

[14] L. Ingber. Very fast simulated re-annealing. *Mathl. Comput. Modelling*, 12(8):967–973, 1989.

[15] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[16] L. Li, A. Martinoli., and Y. Abu-Mostafa. Emergent specialization in swarm systems. In H. Yin, N. Allinson, R. Freeman, J. Keane, and S. Hubbard, editors, *Proc. of the Third Int. Conf. on Intelligent Data Engineering and Automated Learning IDEAL-02*, volume 2412 of *Lecture Notes in Computer Science*, pages 261–266. Springer Verlag, August 2002.

[17] A. Pikovsky, R. Rosenblum, and J. Kurths. *Synchronization, A universal concept in nonlinear sciences*, volume 12 of *Cambridge Nonlinear Science Series*. Cambridge University Press, Cambridge, UK, 2001.

[18] J. Pollack, B. Jordan, H. Lipson, S.G. Hornby, and P. Funes. Three generations of automatically designed robots. In *Artificial Life VII Proceedings*.

[19] D. Rus, Z. Butler, K. Kotay, and M. Vona. Self-reconfiguring robots. *Communications of the ACM*, 45(3):39–45, 2002.

[20] B. Salemi, W.-M. Shen, and P. Will. Hormone controlled metamorphic robots. In *Proc. of Intl. Conf. on Robotics and Automation*, May 2001.

[21] W.-M. Shen, Y. Lu, and P. Will. Hormone-based control for self-reconfigurable robots. In *Proceedings of the fourth international conference on Autonomous agents*, pages 1–8. ACM Press, 2000.

[22] W.-M. Shen, B. Salemi, and P. Will. Hormone for self-reconfigurable robots. In *Proc. Intl. Conf. Intelligent Autonomous Systems*, pages 918–925. IOS Press, 2000.

[23] K. Sims. Evolving virtual creatures. In *Siggraph '94 Prodeedings*, pages 15–22, July 1994.

[24] sims94b. Evolving 3d morphology and behavior by competition. In Brooks and Maes, editors, *Artificial Live IV Proceedings*, pages 28–39. MIT Press, 1994.

[25] P. Will, A. Castaño, and W.-M. Shen. Robot modularity for self-reconfiguration. In *Proc. SPIE Sensor Fusion and Decentralized Control II*, pages 236–245, Boston, September 1999.

[26] Y. Yang, K. Roufas, C. Eldershaw, M. Yim, and D. G. Duff. Sensor computation in modular self-reconfigurable robots. In Bruno Siciliano, editor, *Experimental Robotics VIII*, Advanced Robotics Series. Springer Verlag, 2003.

[27] M. Yim, K. Roufas, D. G. Duff, Y. Zhang, C. Eldershaw, and S. Homans. Modular reconfigurable robots in space applications. *Autonomous Robot Journal, special issue for Robots in Space*, 2003.

[28] M. Yim, Y. Zhang, K. Roufas, and D. G. Duff. Connecting and disconnecting for chain self-reconfiguration with polybot. *IEEE/ASME Transactions on mechatronics, special issue on Information Technology in Mechatronics*, 2003.

[29] Y. Zhang, M. P.J. Fromherz, L. S. Crawford, and Y. Shang. A general constraint-based control framework with examples in modular self-reconfigurable robots. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, Lausanne, Switzerland, October 2002.

[30] Y. Zhang, M. Yim, C. Edlershaw, K. Roufas, and D. Duff. Attribute/service model: Design patterns for efficient coordination of distributed sensors, actuators and tasks in embedded systems. In *Workshop on Embedded System Codesigns*, 2002.