BIOROB
EPFL Biorobotics Laboratory

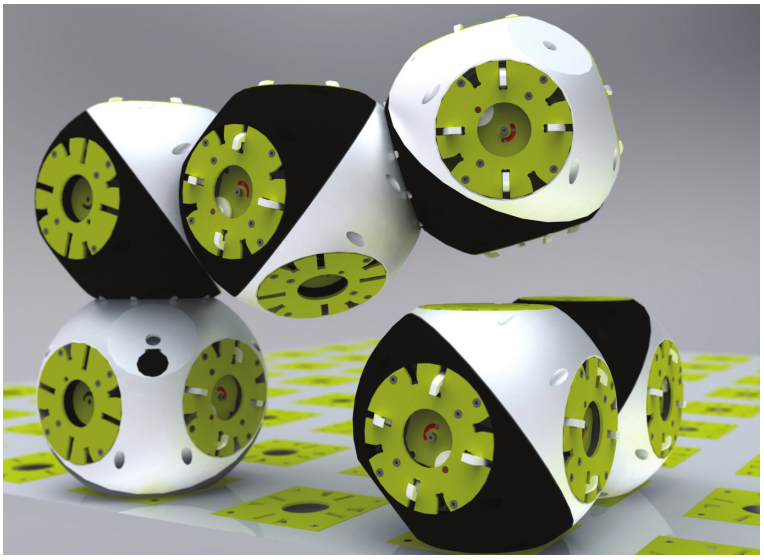
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Spring 2009-2010, IC-SIN

Start: 22.2.2010
Finish: 25.06.2010

Co-evolution of Morphology and Control for Roombots

Ebru Aydın



Professor:	Auke Jan Ijspeert
Assistants:	Rico Möckel, Jesse van den Kieboom , Soha Pouya, Alexander Spröwitz

Acknowledgements

I would like to express profound gratitude to Prof. Auke Ijspeert, for giving me the opportunity of working on such an interesting project. I would also like to thank Rico Möckel, Jesse van den Kieboom , Soha Pouya and Alexander Spröwitz for their invaluable assistance, motivation and guidance.

I also express my thanks to my friends, who never left me alone whether they are near or far, for emotional support, amity, entertainment, and caring they provided.

Last and foremost, I would like to thank to my family, my parents Nuray and Niyazi and my grandmother, for their endless love and support. I would like to thank to my sisters Burcu and Duygu, for their guidance, trust and encouragement. Finally, I express my profound appreciation to Göl for his support, understanding, and patience.

Abstract

The general approach in robotics is to design the robot hardware, then the controller of the robot for the given task. In case of the modular robots, generally the controllers for the hand designed structures are optimized even though they can assemble into many different structures.

This master project presents a co-evolutionary approach to design robots composed of the Roombots modules and additional passive elements. The encoding method which is based on L-systems and inspired from biological developmental process combines the controller and the morphology. A language composed of build commands for the Roombots is used for the L-systems.

Fast, energy efficient and safe locomotion is used as the desired performance of the evolution in the experiments. The proposed system is capable of integrating with further building blocks. Passive telescopic legs are added to the system. A variety of gaits for the Roombots are explored with and without passive legs. Faster gaits are evolved with passive legs.

Contents

Chapter 1 Introduction

Developing a robot consists of two main steps: developing a hardware and designing the controller. In many robotics project, first the hardware is designed according to a given task, and then the controller is designed for that hardware. It is already known that it is not trivial to design an optimal controller to a given hardware. Co-evolution for robotics is also known as body-brain evolution, which means evolving the morphology and the controller of the robot together. Co-evolutionary methods try to find the best solution for the given task by optimizing physical structure and controller together. Robotics researchers used a variety of components as the building blocks of the evolution.

Sims (1994b) evolved robots from cuboids and artificial neurons in simulation. In the Framsticks project Komosinski (2000) and in the Golem project Lipson and Pollack (2000) evolved creatures from sticks and artificial neurons. These research works show that creative and efficient creatures can be constructed for a given task by using simple structures as morphological building blocks. This project aims at exploring the benefits of using co-evolution for modular robots. Modular robots, generally, are simple units designed to build up more complex structures.

This research work covers developing a co-evolutionary system for modular robots. The goal of the evolution is to find fast, energy efficient and safe -less probable to hit the ground- gaits by evolving the controller and the morphology together. Moreover, the effects of using passive compliant elements on the defined fitness criteria is also investigated. Roombots, which explained in detail in next sections, are used as the morphological building blocks. The Roombots are designed as uniform modular robots which are capable of attaching and detaching to each other, and moving together.

The unification of the controller and the morphology of the robot is required for co-evolution. Therefore a coupled network of oscillators is employed as the controller which has structural similarities with the Roombots. To understand the method derived in this research work, information on genetic algorithms, genotype encoding and the Roombots is required.

1.1 Genetic Algorithms

Genetic Algorithms (GAs) are adaptive heuristic search algorithms inspired from the evolutionary ideas of natural selection and genetic. The algorithm starts with a set of solutions. Based on Darwin's "Survival of the fittest" theory, the fitness of the solution determines its reproductive success (Floreano & Mattiussi, 2008). Mutation and crossover are widely used operators in GAs. The solutions are described with genotypes and the operators are defined on these. Phenotype means the solutions itself, in this case the robots together with their controllers compose the phenotype. In the following section the encoding types which are widely used in evolutionary robotics are given.

1.2 Genotype Encoding

Genotype encoding allows genotypes to describe concisely complex phenotypes (Komosinski & Rotaru-Varga, 2002). Komosinski and Rotaru-Varga (2002) stated that encoding is the most important element of the genotype-fitness relationship and the performance of the evolution is significantly affected by the encoding. Floreano and Mattiussi (2008) stated the representation should not limit the number of possible solutions and it should be well tailored for the problem. Longer genotypes correspond to larger search spaces which reduce the probability of producing improvement, as they mentioned the encoding has a critical effect on the evolution. Therefore, the encodings in the literature are examined before adoption. A variety of encoding types to represent the solutions in evolutionary robotics are researched in the literature. Some of the encoding types used in evolutionary robotics are briefly explained.

Direct encoding: In this method, direct specifications of the solution are given in the genome. Generally single point mutations are defined and crossover is not straight forward. Use of repair operations are common for this encoding method after evolution operations. Lipson and Pollack (2000) and Endo, Maeno, and Kitano (2002) used direct encoding to represent the morphology and controller of the robots.

Direct Recurrent Encoding: A compact direct encoding method which has a one-to-one correspondence between genotype and phenotype, is designed for Framsticks project (Komosinski & Rotaru-Varga, 2002) (Komosinski, 2000). This encoding is designed to make the genotypes compact and robust in the face of genetic operators.

Cellular Encoding: In this method the solutions are represented as a set of directions for constructing the solution, rather than as a direct specification.

Generative (Developmental) Encoding: Developmental encoding mod-

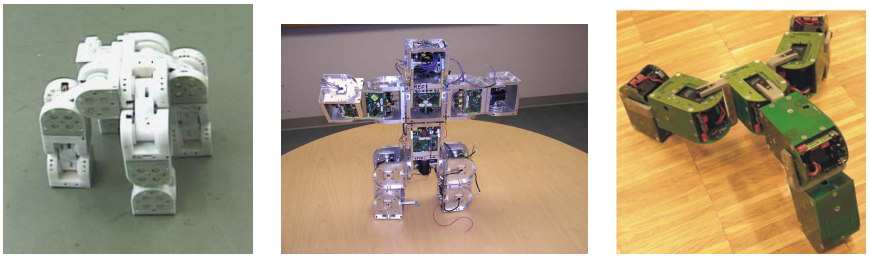
els biological growth of the individuals. It allows reuse of components like sub-procedure calls. Graph based (Sims, 1994b) (Sims, 1994a) (Marbach & Ijspeert, 2004) and grammar based(Hornby & Pollack, 2001b) (Hornby, Lipson, & Pollack, 2001) (Pollack, Lipson, Hornby, & Funes, 2001) developmental encoding methods are used to generate robot structures and their controllers in the literature. It is stated on most of these projects that developmental encoding brings self-similar structures and compact representation. Additionally Floreano and Mattiussi (2008) stated that developmental representation permits and favors modularity and symmetry. Also they mentioned it brings scalability since developmental process is a self-organized and distributed process characterized by parallel operation. Comparative experiments showed that generative encoding creates fitter creatures and structures in a faster way than non-generative encoding for the test environment (Hornby & Pollack, 2001b) (Hornby et al., 2001) (Hornby & Pollack, 2001a).

In this research work the genotypes are encoded using developmental encoding considering aforementioned properties and its intuitive support of modularity. Specifically, Lindenmayer systems (L-systems) are used to encode the Roombots structures and the controllers, more information about these systems and the encoding method is given in Chapter 2.

1.3 Roombots

The Roombots are modular robots that are designed to construct adaptive and self-configuring furniture at BioRob. Instead of having dedicated robot designs, the ability to use the robots in different situations inspired many researchers. A few examples of modular robots from literature are given In Figure 1.1. "Modular Transformers" (MTRAN) is a general purpose modular robot, MTRAN is able to change its shape without human interventions (self-reconfiguration) and make various motions (Kurokawa, Yoshida, Tomita, Kokaji, & S. Murata, 2003). SuperBot is another self-reconfigurable robot with three degrees of freedom (DOF). Salemi, Moll, and Shen (2006), mentioned that even though the fixed robots would perform better on the fixed tasks, self-reconfigurable robots may outperform the fixed one via adaptation to the environment. Sproewitz, Moeckel, Maye, Asadpour, and Ijspeert (2007) showed that using central pattern generators (CPGs) to control the YaMor modules provides robustness in imperfect conditions.

The Roombots are self-reconfigurable, self sufficient and uniform modular robots. Each Roombots module is equipped with a battery, a communication unit, an actuator and connection units, thus they are able to work autonomously. A Roombots module can be seen in Figure 1.2(b). It is composed of four half spheres. The size of a module is $220mm$ by $110mm$ by



(a) MTRAN (b) Superbot (c) YAMOR

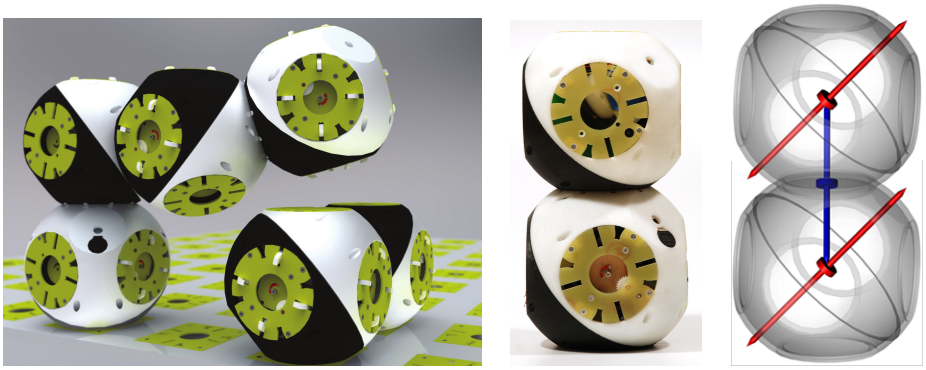
Figure 1.1: Modular robot examples. (a) MTRAN is a self-reconfigurable modular robotic system with one DOF. It is able to make various 3D shapes and motions (Kurokawa et al., 2003). (b) SuperBot is a multi-functional self-reconfigurable modular robotic system designed for NASA space exploration programs (Salemi et al., 2006). (c) YaMor is a self-sufficient modular robot in terms of actuation, actuation control, energy supply, processing and communication (Sproewitz et al., 2007). The images are taken from the cited articles.

$110mm$ and one module weights $1.4kg$. The yellow circles on these spheres correspond to connection ports. In Figure 1.2(b) there is an active connection mechanism(ACM) on the lower sphere and a passive one on the upper sphere. The gearbox is visible on the ACM. Each module has three rotational degrees of freedom, axes of rotations are represented in Figure 1.2(c) and each of these joints is able to rotate continuously. The outer joints which are shown with red axes in Figure 1.2(c) make a full round in two seconds and the inner one in three seconds (Spröwitz et al., 2010). Spröwitz et al. (2010) optimized control parameters for locomotion of two connected Roombots modules which is called "metamodule" and control parameters for a quadruped structure in simulation. These results are important for this project too, since they supply the basic data by means of Roombots' locomotive ability.

1.4 State of the Art

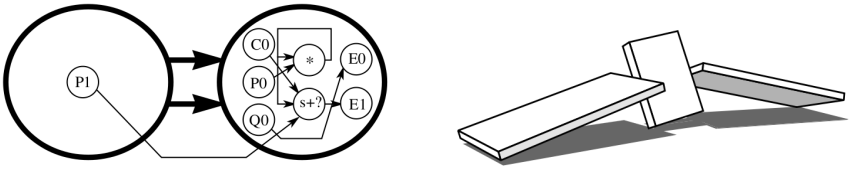
Sims (1994b) (Sims, 1994a) designed a framework to evolve 3D structures in simulation. He evolved both morphology and controller for the creatures using tree based generative encoding. This study can be seen as a milestone on co-evolutionary robotics. Most of the other projects referred it. In Figure 1.3 an example encoding and corresponding creature are given, in the graph nodes neural network architecture and parameters are stored for this part of the body.

Lipson and Pollack (2000) used a co-evolutionary approach to evolve the robots. The developed system offers full autonomy at the levels of power



(a) Roombots rendered (b) A real Roombots module (c) Three axes of rotation

Figure 1.2: Three rendered Roombots modules are given in Figure (a). (b) shows a real Roombots module with one active (lower) and one passive (upper) connection mechanism. In (c) axes of DOfs are given. Each Roombots module has 3 DOF, any of these three joints are continuously rotational (Spröwitz et al., 2010).



(a) Genotype graph (b) Corresponding creature

Figure 1.3: An evolved creature. (a) shows the genotype of the creature. A graph node corresponds to a cuboid in the phenotype. (b) shows phenotype of the evolved creature. The node at the left side in (a) contains control parameters and structural information about the central cuboid and node at the right side in (a) stores same information for the arm like structures. Images are taken from (Sims, 1994a)

and behavior as well as at the levels of design and fabrication. It is able to manufacture the evolved robots by using 3D printing technology right after the the evolution in an automatic way. Elementary building blocks include bars and actuators for morphology and artificial neurons for neural network controller. Creatures are represented using direct encoding, a string of numbers that describe bars, neurons and their connectivity used as genome and speed measured as fitness function. An evolved creature is given in Figure 1.4.

Endo et al. (2002) used co-evolution for biped locomotion and compared

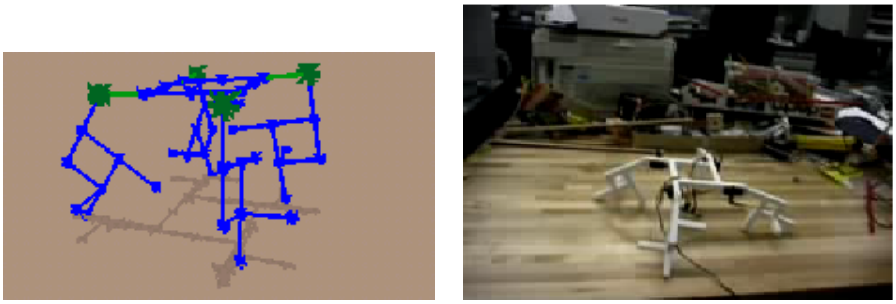
(a) Virtual (b) Manufactured

Figure 1.4: Arrow like creature, (a) in simulation and (b) shows the manufactured robot. The robot has three actuators. One of them is in the center and two of them are in the side limbs. The robot has symmetric structure and controller. The side actuators are synchronized and work in anti-phase with the central one. Images are taken from http://demo.cs.brandeis.edu

the velocity of the evolved robot to the velocity of a hand designed biped robot controlled with an evolved neural network. In this work, direct encoding on a fixed length genome is used for solution representation. A body structure with hip, knee, ankle joints is used. In one part of the project geometry of the parts between joints (morphology) and the neural network parameters are evolved, in the other part neural network parameters for an existing robot are evolved. The experiments showed that evolving morphology and the controller together yields better results than evolving controller for a hand designed robot. This project is different than many other co-evolution projects, because Endo et al. (2002) considered building actual practical robots together with the mechanical aspects.

In 3D Genobots project Hornby and Pollack (2001b) used Lindenmayer systems (L-systems) as the common generative encoding for both body and brain. A detailed description of these systems are given in the next chapter. Morphology and controller build commands are combined in the encoding. Whenever an actuator command is encountered an output is taken from the neural network, by this way body and brain are combined and evolved together. The results are compared to non-generative encoding and it is concluded that generative encoding produced faster creatures with greater self similarity. Figure 1.5 shows an evolved creature where morphological building blocks such as sticks and actuators, can be seen.

Daniel Marbach, a former student of BioRob, also worked on co-evolution for modular robots. He used tree based representation for controllers and configurations for Adam, a modular robot simulation and evolution tool (Marbach & Ijspeert, 2004). Marbach and Ijspeert (2004) concluded that with the co-evolutionary approach efficient and creative solutions are discovered. He also stated that successful individuals are tend to be symmetric even though it was not explicitly rewarded.



(a) Virtual (b) Real

Figure 1.5: An evolved symmetric creature from (Hornby & Pollack, 2001b). Generative encoding is used to represent the morphology and controller. (a) Green marks show actuated joints and blue is drawn for the sticks. (b) The manufactured version of the same creature. Images are taken from http://demo.cs.brandeis.edu

1.5 Scope of the Study

Modular robots can attach to each other with various numbers and shapes. They can construct many different robots. In classical approach first the structure constructed by the modular robots is determined, then the optimum controller is designed. However, this approach covers a limited number of possible configurations of the modular robots. On the other hand, co-evolution optimizes the controller while developing the structure; in this way, the proposed method can adapt the controller to various shapes of the structure. There is a high variety of the shapes and controllers that can be developed, if co-evolution is employed. Throughout this research work locomotion is employed as the desired task, but the proposed method can also be applied to any other task.

The proposed encoding method which combines the representation for the Roombots structures and the controllers is designed based on the L-system grammars and inspired from biological developmental processes. An evolutionary algorithm is developed and implemented for this specific encoding. The solutions of the algorithm are evaluated in simulation according to their locomotive ability, their energy efficiency and impact values they received from the ground. In order to simulate them, a system is implemented to decode the solutions and to prepare simulation files for the robots. Finally two different sets of experiments are performed. First the proposed system is tested using only the Roombots modules as the morphological building blocks. Then passive compliant elements are added as building blocks and their effects on the evolution criteria are investigated

This work is structured as follows: Chapter 2 contains background information about L-systems. It also explains the developed encoding method

and the evolutionary algorithm. An example is given to explain how the Roombots structures are evolved and built. Chapter 3 provides implementation of the build and testing system. Chapter 4 presents the experimental results and analyzes them. Finally the report is ended with conclusion.

Chapter 2 Evolving Robots

This chapter explains the proposed co-evolutionary system and the encoding strategy in detail. Firstly, the Lindenmayer system concept is introduced, since the robots are encoded using L-systems. Then, requirements to encode a robot is given before explaining the proposed encoding strategy. Finally, the details of the method with an example and the designed evolutionary algorithm is given.

2.1 L-System

L-systems are initially developed to model the growth process of the organisms (Floreano & Mattiussi, 2008). In this thesis, the L-systems are used to encode the construction process of the robots and their controllers. L-systems are also known as rewriting systems. In these systems starting from the start symbol, the string is rewritten according to rewrite rules in a parallel way. In other words every symbol will be replaced at each step at the same time. There are many types of L-systems such as parametric, stochastic, deterministic, context-free, bracketed, etc. An L-system is a variant of formal grammar, thus it is defined with a tuple G (Floreano & Mattiussi, 2008). V denotes the alphabet, which is a set of symbols. The symbols which can be replaced are called non-terminal symbols or variables and the others, which are assumed to be replaced according to identity production $(a \rightarrow a)$, are called terminal symbols or constants. w is called axiom which defines the initial state of the system, and is also known as the start rule. P is set of rules, defines how the non-terminal symbols will be replaced. A formal definition of the grammar is given below.

$$G = (V, w, P)$$

$$w \in V^{+}$$

$$P \subset V \times V^{*}$$

Lindenmayer modeled growth process of various types of algae using a deterministic context-free L-system. The original L-System developed by Lindenmayer to model algae is given below (Prusinkiewicz & Lindenmayer,

1990).

$$V = \{a, b\}$$
$$w = a$$
$$P = a \rightarrow ab, b \rightarrow a$$

Starting from axiom, a, the following sequence is generated.
Step0 : a
Step1 : ab
Step2 : aba
Step3 : $abaab$
Step4 : $abaababa$
Step5 : $abaababaabaab$
Step6 : $abaababaabaababaababa$

In Figure 2.1, another L-system is given. This system has only one rule. At every rewrite step each line segment is replaced with the generator which is given in Figure 2.1. Such graphical representations are also used to model the plants, however to model higher plants more sophisticated models are required, for this purpose turtle representation is designed. After the symbols are rewritten they are interpreted as move commands of a turtle. The State of a turtle defined with its position and heading. The grammar given below also generates snow flakes when the string is interpreted using turtle representation. Whenever an F is encountered the turtle moves forward, for $+, -$ it changes its heading. The turtle rules based on only these three operators always produce a line.

$$V = \{F, +, -\}$$
$$w = F - -F - -F$$
$$P = F \rightarrow F + F - -F + F$$

The L-systems are created to model tree-like structures, therefore a representation to enable branching is required. The bracketed L-systems support such structures via bracket operators ([ ]) which are used to save and restore the current state of the turtle. "[" saves the state to the stack and "]" pops the state from the stack.

As mentioned in the previous chapter, there are many robotics projects that used L-systems as generative functions of the robots in the literature. In these systems terminal symbols are designed to be build commands of the morphology and the controller, Hornby and Pollack (2001b) used turtle rules to create the robot morphology using sticks as building blocks. A stick

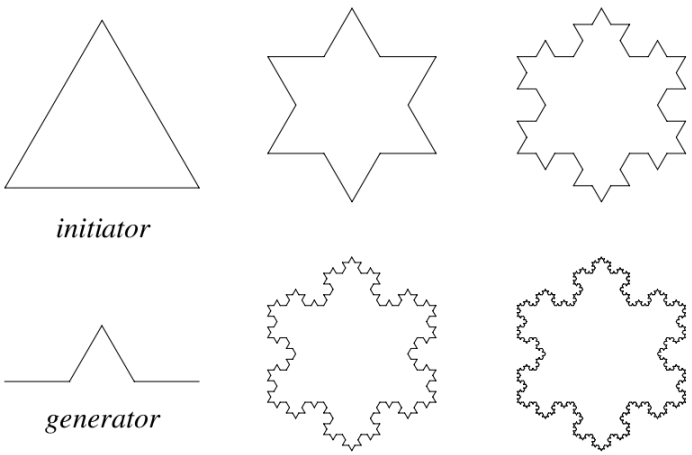


Figure 2.1: L-system to generate snow flakes. Reproduction starts with the initiator. At every rewrite step each line segment is replaced with the generator.Image is taken from (Prusinkiewicz & Lindenmayer, 1990)

is added to the robot as the turtle moves. In these projects, build commands are set in a way that they either add a simple part to the morphology or they modify a control parameter. The main difference between previous projects and this project lies here. In this project the modular robots are used as morphological building blocks. The Roombots are not simple compared to the actuated sticks or cuboids in terms of control (3 DOF) and structural properties. Moreover, the shapes of the Roombots are fixed, only the configurations are evolved as in (Marbach & Ijspeert, 2004).

Using the Roombots brings many constraints. The number of modules that can be used to build a robot is limited. One limitation is that a module can lift up to two modules including itself. Furthermore increasing the number of modules decelerates the simulation excessively (Experimentally it is observed that the simulation time increases exponentially). Consequently, possible evolved robot structures are limited compared to other projects. Furthermore, the physical properties of the modular robots have to be defined in the language, in the Roombots case there are many connection configuration settings which has to be specified whenever a module is added during the construction. Adding a new module gives three new DOF whose parameters have to be set, and results in a more complex language.

The design of the the language which is the set of terminal symbols, in other words the build commands, is an important part of the project. The language has to be able to build all possible robot configurations and controllers, thus it has to cover the search space for CPG controlled Roombots configurations. Moreover, redundancy should be avoided, even with the most optimum representation the search space is large, so additional

redundancy would lead to unsuccessful searches.

2.2 Controller

CPGs are neural networks that can produce rhythmic patterned outputs without rhythmic sensory or central input (A. J. Ijspeert, 2008) (Righetti & Ijspeert, 2006). A. J. Ijspeert (2008) reviewed the CPGs for locomotion control in animals and robots. CPGs are proven to be a successful control method for robots in many projects. The method is inspired from biology, neuro biological experiments showed that CPGs are distributed networks made of multiple coupled oscillatory centers (A. J. Ijspeert, 2008). A CPG based controller brings advantages. First of all exhibiting limit cycle behavior provides robustness, it is well suited to distributed implementation, it reduces dimensionality of the problem and it is suitable for optimization algorithms (A. J. Ijspeert, 2008). Distributed implementation is essential for modular robotics. Additionally it is important for optimization. It not only well parameterizes the solutions but it is also suitable for randomly generated robot configurations. Identical local control units are used for each module and these units are connected via couplings.

In the proposed system an oscillator is used for each DOF. The servo motors in the spheres of a module are called $s1$ and $s2$ and the servo motor between the spheres of a module is called $m1$. The oscillators that control the servo motors are coupled in a module as shown in Figure 2.2 and physically connected modules are coupled through their $m1$ servo motor oscillator regardless of their connection faces and types. Bi-directional coupling is used for intra and inter module couplings.

The servos are governed by the following equations, this controller architecture is designed at BioRob for the Roombots (Spröwitz et al., 2010). Each servo is associated with one oscillator and equations for an oscillator are given below.

$$\dot{\phi}_i = 2 \cdot \pi \cdot \nu + \sum \omega_{ij} \cdot r_j \cdot \sin(\phi_j - \phi_i - \psi_{ij}) \tag{2.1}$$

$$\dot{r}_i = a(R_i - r_i) \tag{2.2}$$

$$\dot{x}_i = b(X_i - x_i) \tag{2.3}$$

$$\begin{aligned} \theta_i &= r_i \cdot \sin(\phi_i) + x_i && Oscillation \\ \theta_i &= \pm\phi_i && Rotation \\ \theta_i &= x_i && Locked \end{aligned} \tag{2.4}$$

Three state variables are defined for each of the oscillator, ϕ_i encodes the phase, r_i encodes the amplitude and x_i encodes the offset of the oscillator. ψ_{ij} is the phase bias of the coupling between i^{th} and j^{th} oscillator. Compared with the original controller an additional state for the offset, x_i, is added to

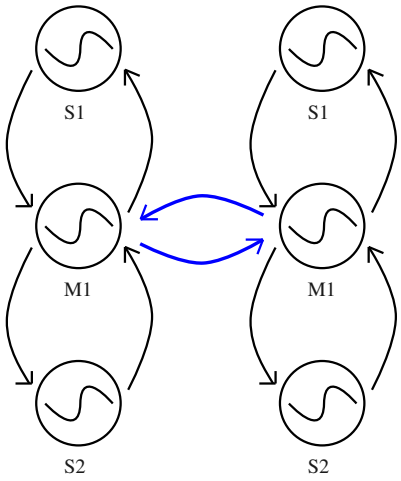


Figure 2.2: The oscillators of two modules, blue links show inter module couplings. Neighboring oscillators are coupled bidirectionally in a module. The centered oscillators of two physically connected modules are coupled in the same way.

the dynamical system. The framework automatically creates the robots and the initial positions of the servo motors are set to zero. This state variable x_i synchronizes the physical state and the controller state.

The frequency of the system ν is a fixed parameter and it is set to 0.25 considering physical properties of the real Roombots. Experiments showed that together with the real Roombots limitations the servos are not able to follow the values generated by the CPG while using higher frequencies. The weight of the couplings(ω_{ij}) are fixed and set to 2 for each coupling. a and b, set to 2, are positive constants which only affect the rise time of the state variables. R_i, ψ_{ij} and X_i are open variables that are subject to optimization. They are set by the build commands which are terminal symbols of the L-system.

The servo driving functions which are given in Equation 2.4 support two types of basic movements, namely oscillation and rotation. Since change of the direction of the rotation can result in different gait, to explore all the possible configurations for joint movement types, both options are considered.

2.3 Morphology

The number of modules used and how they are connected to each other determine the robot structure. Additionally, offsets of the oscillators affect the robot shape. Each connection can be defined with two modules involved, their connection faces and connection type. Figure 2.3 shows connection types. Each module has 10 connection faces, two of them are on the inner spheres and three of them are on the outer spheres. Connection faces are shown with the yellow circles on the Roombots. Each hemisphere features only one active connector on the real hardware. Four ACMs are used in the experiments to be realistic and to limit the number of possible choices. Whenever a new module is added during the construction phase, the connection faces of each module and connection type has to be specified.

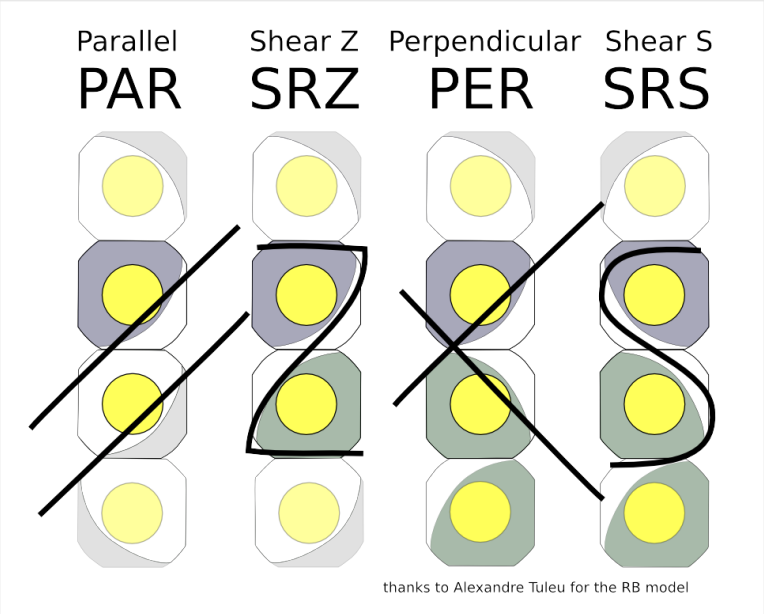


Figure 2.3: Two Roombots modules can be connected in four different ways. These are shown in the figure. Image is taken from the work of Mayer (2009)

Passive Compliant Elements

Passive compliant elements do not require additional control or energy. Consequently, they might bring more efficient gaits without adding extra complexity to the system. Moreover, they might provide safer gaits by reducing the contact of the modules with the ground. The passive compliant elements used in this work are controlled with a spring-damper system. Two types of passive elements are tested. They are either added between modules or attached to only one module. The passive elements between modules

either move rotationally or they change their lengths by compression and stretch (telescopic motion). The leg like passive elements, which have only one connection mechanism, support telescopic movements. In Figure 2.4 a Roombots module and three leg like passive elements are shown. These elements are called passive legs at the rest of the report. They are explained in more detail in the Chapter 4. The passive elements has no control parameters. However, position, length and the mechanical properties are open parameters subject to evolution.

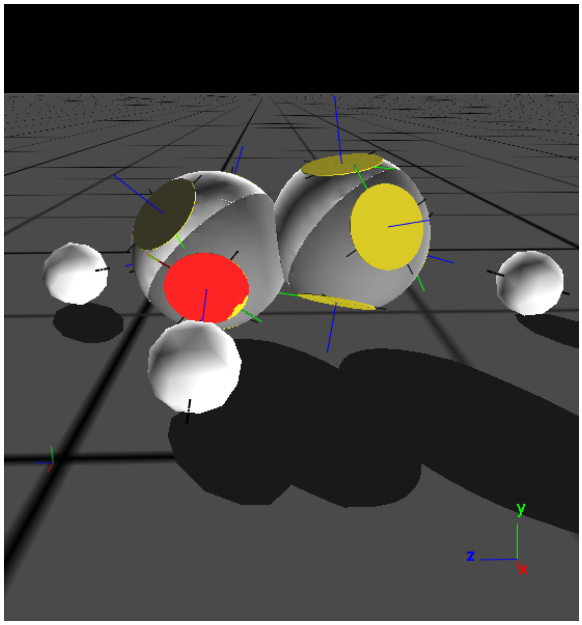


Figure 2.4: A Roombots module with three passive legs. Passive legs are rendered with a red connection face and a sphere.

2.4 The Language

In the proposed language, build commands for the controller and the morphology are combined. The build commands are designed in an additive way. The commands either add a new element, change parameter or modify the state of the system. Interpretation of the rewritten string is done in a similar way as for the turtle interpretation. This way of interpretation enables production of self similar parts. The production starts with an initial module. Then commands in the rewritten string are processed respectively. When a module is added during construction, the connection faces on each module and connection type have to be specified. These define the morphology. Drive function, amplitude, phase offset for each oscillator on new module and phase biases as controller parameters have to be set. Hence to-

Table 2.1: Parameters

Configuration	**Count**
Connection face	2
Connection type	1
Controller	1
Phase bias between modules (Angle)	1
Phase bias in module (Angle)	2
Oscillator amplitude	3
Oscillator drive function	3
Oscillator offset	3

Table 2.2: Parameter domains

Connection face	C0X, C1Y, C2Y, C3X
Connection type	PAR, SRZ, SRS, PER
Oscillator	s1, m1, s2
Drive functions	Oscillatory, Rotational, Locked
Coupling	s1::m1, m1::s2
Angle	$[-\pi, \pi]$, step 10 degrees
Amplitude	[0.5, 3] , step 0.1

tal number of variables which have to specified for every new module is 16. Table 2.1 gives list of these parameters and Table 2.2 presents domains for the parameters. Some of the parameters have continuous domains such as angles. These are discretized for the evolutionary algorithm, the step values for these parameters are also given in Table 2.2. Some of these parameters are designed to be state variable and they are updated using "CHANGE" commands and some of them are specified with "ADD" command. The distinction is done in order to enable production of regular controllers and configurations taking into account the affects of the parameters.

2.4.1 State

The state of the L-system is composed of a module, one connection face of this module and a phase bias. Whenever a new module is added, these three specify the properties related to the previous module. In the rest of the report the module specified in the state will be called as active module and similarly the face will be called as active face. Whenever an "ADD" command is encountered a new module is added to the active module's active face and the active module is set as the new module and one of its face is set as the new active face.

The passive legs are added to the active module without changing the state. The passive elements between modules compose a special case. Since

another module can be added to these, they become the active module. They are not controlled by oscillators, hence $m1$ oscillator of the new module is coupled with the previous one.

The commands starting with "CHANGE_" affect the active module and update one of the settings using the following equation.

"CHANGE_VALUE N" command is interpreted as:

$$VALUE \equiv VALUE + N \pmod{S} \tag{2.5}$$

S is the size of the parameter domain.

2.4.2 Build Commands

ADD *CONNECTION TYPE, CONNECTION FACE*

The "ADD" command adds a new module. It has two parameters which are connection face and connection type. Connection face specifies the connection face on the new module. Controller parameters of the new module are set to default values and they are changed with other commands.

CHANGE_FACE *N*

The "CHANGE_FACE" command updates the active face. This command can be interpreted as the change direction command of the turtle rules. It changes the active face using the following equation which is derived from Equation 2.5:

$$ACTIVE\_FACE \equiv ACTIVE\_FACE + N \quad (mod 4)$$

CHANGE_PHASE_BIAS_BM *N*

"CHANGE_PHASE_BIAS_BM" command updates the phase bias between modules which is saved in the state. There is another command which is defined to update the phase bias between oscillators of a module. The distinction is made to keep the canon since this one changes a state variable and the other one changes an oscillator parameter. Therefore, it is changed in a similar way to other oscillator parameters.

CHANGE_AMPLITUDE *OSC N*
CHANGE_OFFSET *OSC N*
CHANGE_DRIVE_FUNCTION *OSC N*
CHANGE_PHASE_BIAS *COUPLING N*

The first three commands given above have two parameters, "OSC" chooses one of the oscillators of the active module and the other one is the increment used to update the current value as in Equation 2.5. The first parameter of the last command chooses a coupling to update in a similar way to the others.

BRANCH OPERATOR : [ ]

In order to enable branching and leg like structures branch operators are used. **"["** saves all of the the state variables to the stack and **"]"** restores them.

REPETITION OPERATOR : { } *N*

Even though the system is able to produce self similar structures by rewrite rules without repetition operator, the operator ({ }) offers a quick way to achieve repeated operations. The operator replicates the commands inside the brackets "N" times.

ADD_PASSIVE_LEG: *LENGTH*
ADD_PASSIVE: *LENGTH SPRING_COEFFICIENT TYPE*

In order to evolve structures with passive elements, these add commands are used. The length and spring coefficient are parameters to optimize for the passive element in between modules and type chooses the motion type. Spring coefficient is fixed for the passive legs because of the stability problems which is discussed further in Chapter 4.

2.5 L-System Grammars

In this work, deterministic context-free L-system grammars are optimized. The grammars encode building process of the robots using the build commands explained in the previous section. Since these systems are deterministic, each grammar corresponds to a unique robot defined with the rewritten string.

A grammar has exactly four rewrite rules. The number of rules at the successor part of the rewrite rule varies but is limited. The rewrite process starts with the last rule which has the largest id. The rules can use only the rules with smaller ids in order to disable recursive grammars. Enabling recursion and stopping the rewrite at some level would be another choice, however this would add extra and unnecessary complexity to the system since it would favor large structures which can not be tested in the simulation.

The random creation process of a grammar starts with the first rule which only includes build commands as terminal symbols since there is no

Table 2.3: Build Commands' Abbreviations

Command	**Abbreviation**
ADD	ADD
CHANGE_FACE	CF
CHANGE_PHASE_BIAS_BM	CPBBM
CHANGE_PHASE_BIAS	CPB
CHANGE_AMPLITUDE	CA
CHANGE_OFFSET	CO
CHANGE_DRIVE_FUNCTION	CDF

rewrite rule with a smaller id. Each rule other then the first one has to include the previous rule at least once, it may also include other rewrite rules. It is forced to use the previous rule to make sure that all of the rules will have an effect on the phenotype. While creating the successor part of the rule, up to six rules are chosen from the terminal symbols (build commands) or previously defined rewrite rules with equal probability.

Some parts of the rewrite rules have no effect on the phenotype of the creature. Since the state variables have an effect on the phenotype when an "ADD" command is issued, changing state variables without adding another module after them would not change the phenotype. An example of such a situation is given at the grammar given in the following. In this grammar the red printed parts have no effect on the phenotype. These parts are not checked and not pruned during the generation of the grammar. They are kept because they might have an effect on phenotype after evolution operations. The idea comes from biology. There are parts in the mammalian genomes whose function could not be defined. They are called "Junk DNA", these sequences have likely unidentified activity or they may have had in the past (Biemont & Vieira, 2006).

Rewrite rules of a grammar is given in the following. The alphabet of the grammar is the previously defined terminal rules (build commands) and rewrite rules (R1, R2, R3, R4). The start rule is *R*4. The commands are shown using first letters as abbreviations which are given in Table 2.3 The grammar encodes a creature with four modules.

R1 → CPB(s1::m1 90) CA(m1 1.4) CO(s2 150) CPB(m1::s2 220) CO(m1 290)
R2 → ADD(C1Y PER) CDF(s1 3) CDF(s2 3) R1 CA(s2 1.4)
R3 → CPB(s1::m1 100) R2 CA(m1 2.5) 1 x { CA(m1 2.4) }
R4 → CDF(m1 2) CO(s2 340) CPBBM(100) R2 R2 R3 CF(2)

The rewrite steps are showed below. The symbols which are added in

the most recent rewrite step are colored and the rewrite rules are underlined for illustration.

step0 : R4

step1 : CDF(m1 2) CO(s2 340) CPBBM(100) R2 R2 R3 CF(2)

step2 : CDF(m1 2) CO(s2 340) CPBBM(100) ADD(C1Y PER) CDF(s1 3) CDF(s2 3) R1 CA(s2 1.4) ADD(C1Y PER) CDF(s1 3) CDF(s2 3) R1 CA(s2 1.4) CPB(s1::m1 100) R2 CA(m1 2.5) 1 x { CA(m1 2.4) } CF(2)

step3 : CDF(m1 2) CO(s2 340) CPBBM(100) ADD(C1Y PER) CDF(s1 3) CDF(s2 3) CPB(s1::m1 90) CA(m1 1.4) CO(s2 150) CPB(m1::s2 220) CO(m1 290) CA(s2 1.4) ADD(C1Y PER) CDF(s1 3) CDF(s2 3) CPB(s1::m1 90) CA(m1 1.4) CO(s2 150) CPB(m1::s2 220) CO(m1 290) CA(s2 1.4) CPB(s1::m1 100) ADD(C1Y PER) CDF(s1 3) CDF(s2 3) R1 CA(s2 1.4) CA(m1 2.5) 1 x { CA(m1 2.4) } CF(2)

step4 : [1] CDF(m1 2) [2] CO(s2 340) [3] CPBBM(100) [4] ADD(C1Y PER) [5] CDF(s1 3) [6] CDF(s2 3) [7] CPB(s1::m1 90) [8] CA(m1 1.4) [9] CO(s2 150) [10] CPB(m1::s2 220) [11] CO(m1 290) [12] CA(s2 1.4) [13] ADD(C1Y PER) [14] CDF(s1 3) [15] CDF(s2 3) [16] CPB(s1::m1 90) [17] CA(m1 1.4) [18] CO(s2 150) [19] CPB(m1::s2 220) [20] CO(m1 290) [21] CA(s2 1.4) [22] CPB(s1::m1 100) [23] ADD(C1Y PER) [24] CDF(s1 3) [25] CDF(s2 3) [26] CPB(s1::m1 90) [27] CA(m1 1.4) [28] CO(s2 150) [29] CPB(m1::s2 220) [30] CO(m1 290) [31] CA(s2 1.4) [32] CA(m1 2.5) 1 x { [33] CA(m1 2.4) } [34] CF(2)

The index values in parenthesis are added to describe the string. 34 build commands represent the robot.

2.6 Interpretation of the Rewritten String

The construction process starts with a module. The control parameters of this module are set to default values, 0 for phase biases and offsets and 1.5 for amplitudes. The first two commands change the parameters for this module. The first command changes drive function of the $m1$ servo motor, hence it rotates with negative phase. The second command modifies the offset of the oscillator which controls the module's $s2$ servo motor. The third command updates a state variable and sets the active phase bias to

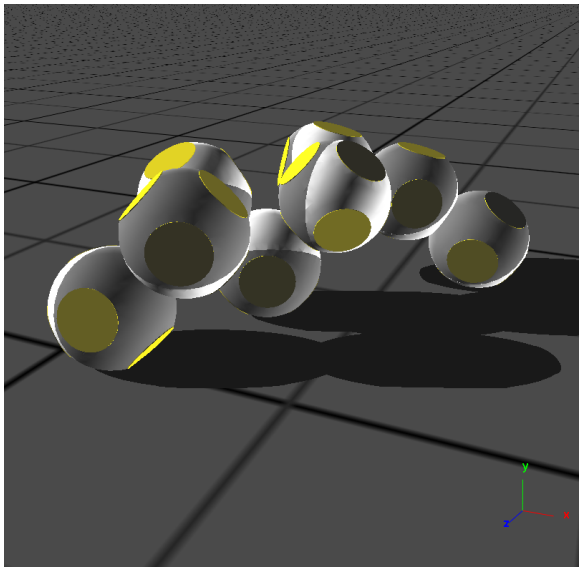


Figure 2.5: The robot which is constructed using the rewritten string shown in the figure. Rightmost module is the first module, construction started with it.

100 degrees. The fourth command adds a new module. Since the active face has not been updated yet, its default value, $C0X$, is used. The new module's $C1Y$ face is connected to the previous module's $C0X$ face in PER style.

The commands between $4-12$, $13-21$ and $23-31$ are rewritten form of $R2$. $R2$ adds one module and modifies its parameters. These three modules' $s1$ and $s2$ servo motors rotate with negative phase $(-p)$. Additionally, phase biases and amplitudes of this module are modified. At 22^{th} point the phase lag between $s1$ and $m1$ servos of the third module is changed. The 32^{th} and 33^{th} commands update amplitude of the $m1$ oscillator of the last module. The previous value of the amplitude was 2.9 since $R2$ also changed it. The 32^{th} command set it to 2.8 and finally the 33^{th} command set it to 2.6

The 34^{th} command updates the active face. It does not effect the phenotype since there is no "ADD" command following it. In Figure 2.5, the robot described above is given. The rightmost module is the first one and the other three are added as described. The Figure 3.2 shows the same robot in its start position.

2.7 Evolutionary Algorithm

An evolutionary algorithm is used to evolve deterministic L-systems. Specifically the rewrite rules of the L-system grammars are evolved. The evolved

grammars are rewritten. The rewritten string is composed of the build commands that are explained in the previous section. Then the robot and their controllers are created and they are evaluated using a physics based simulator called Webots (Michel, 2004) according to their power usage, impact from ground and speed. Evolution then proceeds with the selection of high fitness individuals.

Initialization is done randomly as explained in the previous section, the grammars are checked if the number of modules are suitable whenever a new grammar is created. As explained before, because of mechanical and environmental reasons the number of modules that can be used in a creature is limited up to eight. The minimum number of modules is set to two. Since locomotive capabilities of a single module are almost non-existent, it is not interesting to explore.

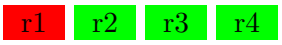
2.7.1 Mutation

Mutation creates a new individual by copying rules from the parent and making small changes on it. Mutations affect the right part, successor, of the rewrite rules. Changes that can occur are insertion/deletion of a rule or perturbing a parameter of one of the build commands.

Each rewrite rule of an individual is mutated with 10% probability. If a rewrite rule is chosen for mutation, a parameter perturbation occurs with 50% probability, insertion and deletion of a rule occur with 25% probability. In case of a new rule insertion, the rule and its position are randomly chosen. In case of deletion again a rule is chosen randomly regardlessly whether it is a build command or a rewrite rule. Similarly for parameter perturbation case the rule and its parameter are chosen randomly and new value is created randomly from its domain.

2.7.2 Crossover

Crossover creates a new individual by copying rules from parents using a randomly generated index. These colored boxes show the new individual after a crossover, the colors of the boxes indicate which parent they came from.



Each individual is crossed over with a randomly chosen individual from the new generation with 20% probability.

2.7.3 Selection

Roulette wheel, fitness proportionate selection, method is used for selection (Floreano & Mattiussi, 2008). The fitness of the solution determines the

selection probability of the individual according to the following function:

$$p_i = \frac{f_i}{\sum_{j=1}^{j=N} f_j} \tag{2.6}$$

where p_i is the probability to chose i^{th} individual for next generation, N is the population size and f_j is the fitness value of the j^{th} individual. Each selected individual is mutated and crossovered with the given probabilities. Moreover, after the selection the populations is checked against duplicates. If two grammars are exactly the same, one of them is replaced with a randomly generated one.

2.7.4 Fitness

Initially fast forward locomotion is set as the task to achieve. Therefore, traveled distance is considered as the fitness function. Speed is calculated using the first and last positions of the robot, to favor locomotion in a straight line. The position of the robots' geometric center is used for speed calculation. However, first experiments showed that using only the speed as the fitness function evolves large structures pushing themselves and roll. A more detailed examination of these creatures showed that they received high impacts from the ground. These high impacts are not tolerable by the real Roombots hardware, since the modules might break. Hence the impact received from the ground is combined into the fitness function. Also it is observed that most of the servos were rotating or oscillating even though they have little or no effect on the gait. In order to evolve energy efficient robots, the torques of the servos are measured and taken into account while calculating the fitness. The following fitness function is used for the experiments:

$$fitness = \frac{speed^a}{\dot{impact}^b \cdot torque^c} \tag{2.7}$$

The modules' spheres are implemented as perfect spheres in the simulation environment. Therefore, a sphere can have at most one contact point with the ground. The impact is calculated as the norm of the forces applied to the sphere at the contact point. Only the average of the positive change of the impact for each sphere, which might damage the module, affects the total impact. In other words, the average of the positive first derivative is calculated for each sphere. The sum of these averages which is shown as $\dot{impact}$ is employed in the fitness function. The *torque* is measured for each servo at each time step, and then averaged. The average torque of each servo of the robot is summed up to calculate *torque*. They might seem favoring smaller structures. However, it keeps the balance between the smaller and larger structures because the larger ones are capable of moving faster. The values for a, b and c are set experimentally respectively 3, 0.5 and 0.5.

Since the creatures are initialized and modified randomly some of them are self-colliding, zero fitness is assigned to such creatures.

Chapter 3 Implementation

There are three main parts in the project to be implemented. These are the optimizer, world builder and simulator. The optimizer sends the solutions to the world builder, world builder constructs the necessary files for simulation then the simulator is invoked and finally the results are sent back to the optimizer from simulator. The framework summary is given in Figure 3.1. Each part is described in detail in this chapter.

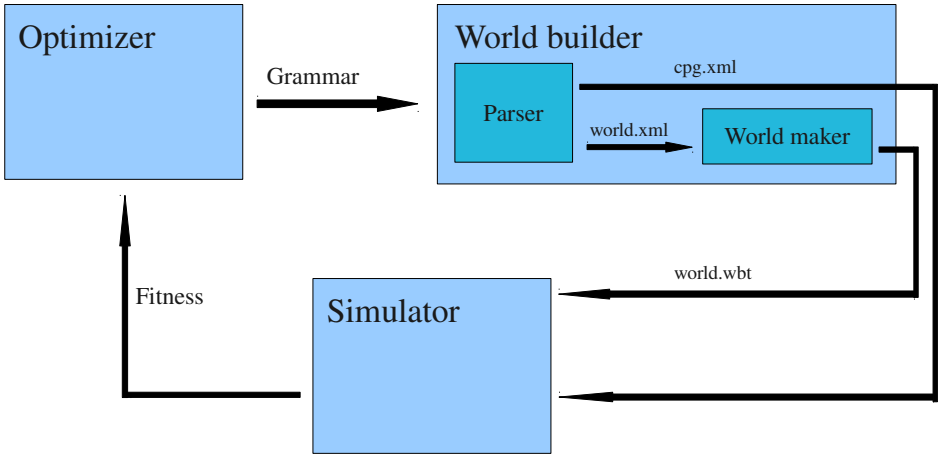


Figure 3.1: The framework has three main parts. The optimizer runs the evolutionary algorithm. The world builder prepares the files describing the creature for simulation. A physics based simulator runs the experiments and sends the results back to the optimizer.

3.1 Optimization

3.1.1 The Optimization Framework

The optimization framework employed in this project, that enables the process to be distributed over many computers, is developed at BioRob (*Optimization Framework Conceptual Overview*, 2010). The user is able to specify the task using an XML file. Since the framework is developed for robotic experiments it has built in support for the Webots simulator. Although the optimization framework supports many optimization algorithms, it was not possible to use them for this project since the solutions which are the L-system grammars can not be coded in a generic way. Moreover the evolution operations are specific to grammars. Hence an optimizer is implemented using the optimization library in *C♯*.

3.1.2 The Optimizer

The evolutionary algorithm explained in Chapter 2 is implemented as the optimizer. The algorithm is able to create and modify L-System grammars. The optimizer always produces syntactically correct grammars. Whenever a build command is added to a rewrite rule, its parameters are also randomly generated. Moreover it keeps track of the parenthesis by using a stack. A randomly produced close parenthesis (}]) is not used unless it matches the last opened parenthesis which is the top element of the stack. At the end of the rewrite rule the rest of the open parenthesis are closed.

Implemented Classes

EA : The EA class derived from the framework's generic optimizer class implements the evolutionary algorithm (*liboptimization-sharp API reference*, 2010).
Grammar : The grammar class derived from the framework's solution class implements the solution of the evolutionary algorithm.
TerminalRules : This class reads the settings file, all functionalities regarding build commands such as generate and update are implemented here.

The grammars are encoded as integer lists and stored in strings. Since the data related to the solutions are saved in a database, the framework obliges to use fixed number of variables for each solution. However, the length of the grammars are not fixed. Also the types of parameters are not uniform, the oscillators are described with strings, the angles are double values and so on. In order to overcome these problems and to save the data in a compact form all of the data is encoded as integers. A text file is used to encode and decode the grammars. A unique number is assigned to each

parameter type and its minimum, maximum and step values are stored. Additionally either set of possible values or a key word to calculate the real value is stored. The following line describes the connection faces:

0 0 3 1 C0X C1Y C2Y C3X

The id of the connection face parameter type is 0, its minimum value is 0, maximum value is 3 and the step value is 1. C0X, C1Y, C2Y and C3X are the keywords describing the Roombots' faces. While decoding the grammar the integer value is used as an index and one of the keywords is returned. The following line describes the "ADD" command:

0 2 0 1

The id of this build command is 0, it requires 2 parameters, and their types are 0 and 1.

This type of encoding/decoding has many advantages. First of all it offers a compact representation. Saving each rewrite rule in a string satisfies the framework constraints since the number of rewrite rules defining a grammar is fixed. Moreover the optimizer part behaves each build command and parameter type uniformly. In other words the optimizer creates and processes the grammars without checking the meanings of the commands or parameters, so it does not use the last part of the parameter types. This implementation helps to change and adapt the build commands easily, and in case of future use of the system it would be easy to manipulate the L-system vocabulary of the system.

3.2 The World Builder

The world builder part is implemented in C++. It receives the grammars via the optimization framework's messaging system. It reads the message and constructs the simulation files. A class named "Individual" is implemented to parse the rules. This class reads the settings file and constructs the "cpg.xml" file and the "world.xml" file accordingly. The "cpg.xml" file stores all of the required data regarding the Roombots' oscillators. The CPG file associated with the robot given at Chapter 2 is given in Appendix A. The formed CPG file conforms to the CPG library specifications (*CPG-Network Reference Manual*, 2010). The CPG library can be used to construct and simulate dynamical systems. The library is used to simulate the oscillators, direct mapping from the library to servo positions is used to control the robot.

A world file in Webots' special format is required to run the experiments. The file represents the world to simulate. Everything which will be simulated has to be defined in this file. More information about Webots' world

files can be found at their website (Webots, n.d.). Since the robot configurations are created randomly, an automatic way of constructing these world files are needed. Thanks to the former BioRob students, there was an application called "WorldMaker" which constructs the world files for the Roombots by using a descriptive XML file. This program is transformed into a C++ class and extra functionalities are added according to the needs of this project. The new version of the WorldMaker is able to add passive elements between modules and at the ends of the modules. The previously mentioned "world.xml" file is passed to this class to write the world files. The XML file defining the robot at Chapter 2 is given in Appendix B.

The configurations of the modules are defined using a tree structure. During the construction process each module or passive element is added after their parents and their positions are calculated accordingly. The WorldMaker is able to set the initial servo positions and calculate child positions respectively. A global rotation matrix can be used to rotate the robot. However, this function is not used. Since the structures are created randomly, their stable positions are not known. The positions are set in the WorldMaker in a way that all of the creatures will be above the ground, so in some cases the robots start in the air or they start in unstable position and they fall to the ground. The consequences of this is handled at the simulator controllers. In Figure 3.2 a robot is given at its initial position.

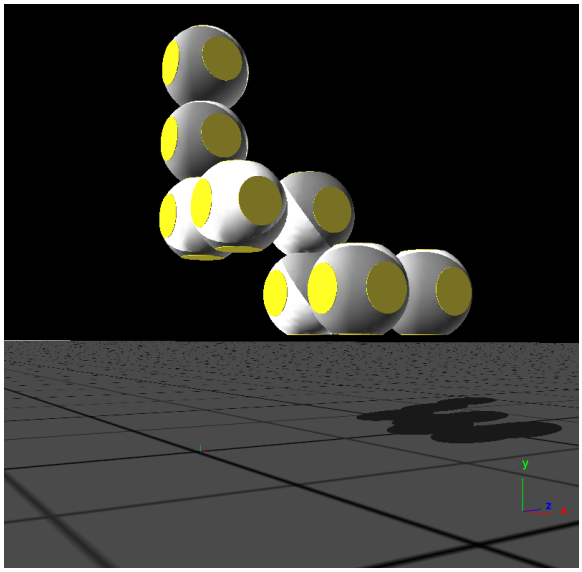


Figure 3.2: A robot is shown at its start position. It starts 22cm above the ground. Initial positions of all the servo motors are set to zero.

3.3 Webots Controller

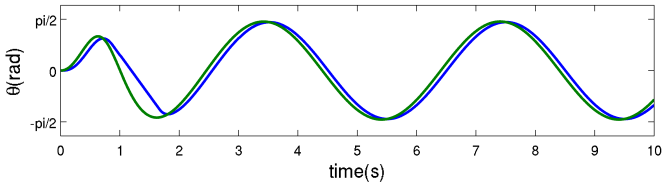
After the creation of the files as explained in the previous sections the Webots simulator is invoked. Each module is associated with a module controller and passive elements with a passive controller. A supervisor controller derived from Webots' supervisor controller is implemented as the central controller. The supervisor controller loads the CPG file and using the CPG library simulates the dynamical system. At each step it sends the servo positions to modules subsequently modules update their positions by using Webots' set-position function which uses a P-controller for position control. In Figure 3.3, the positions sent and the actual servo positions are given. As can be seen from the figure all of the servos are synchronized within two seconds.

The supervisor also checks the collisions by receiving position information from the modules and passive elements. It terminates the simulation with zero fitness in case of a collision. A GPS receiver is positioned to centers of the spheres of the modules and to the center of the passive leg sphere. Since all objects are assumed to have spherical shapes, the collision checks are done using distances between centers of the spheres. The collision detection for the passive elements in between modules are implemented using point to line distance equations.

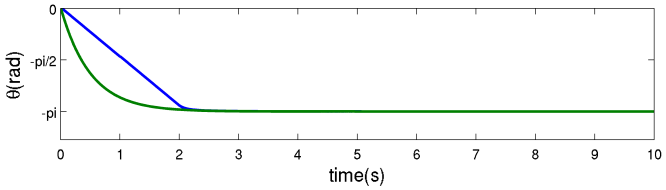
As shown in Figure 3.3, the oscillator network synchronizes in two seconds. The synchronization time depends on the robot size, however it generally synchronizes in less than three seconds. At third second of the simulation a Webots function is called to reset the simulation. This functions stops the inertia of all solids in the world, consequently it stops all solids in the world. Simulation reset banishes the effects of the initial fall. At the fifth and 25^{th} second the supervisor calculates the position of the geometric center of the structure to measure the speed of the robot. During this time, the optimizer receives the impact values from the physics plug-in and servo torques from the modules. At the end of the simulation it sends the related data to the optimizer.

The passive elements are not actually controlled, the controllers are only needed to read GPS values and to send them to the supervisor. The supervisor can not read them according to Webots regulations.

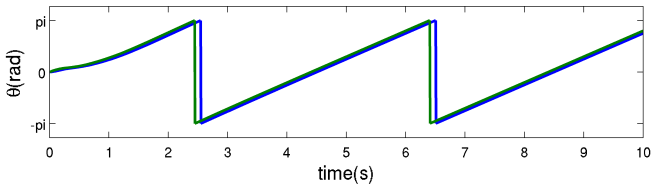
The module controllers receive servo positions from the supervisor and set them to the corresponding servo motors. They read the torque of the each servo and positions from GPS devices and send the information of torque and position to the supervisor.



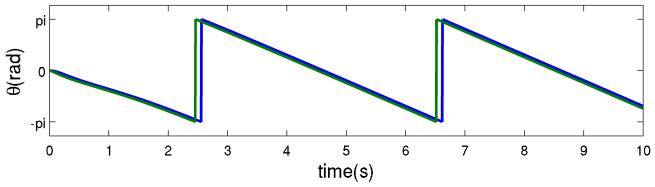
(a)



(b)



(c)



(d)

Figure 3.3: Blue lines show the servo positions sent by the supervisor and the green lines are the actual servo positions. In (a) Positions of an oscillating servo with 1.5 amplitude is given. (b) Servo is locked at $-\pi$ *rad*. (c) (d) Rotating servos with opposite directions

Chapter 4 Experiments and Discussion

4.1 Experimental Setup

The system introduced in the Implementation chapter is used for the experiments. First experiments are performed to tune the parameters. The number of the individuals in the population is fixed to 100. The experiments were run up to 500^{th} generation. This upper limit was set experimentally. The experiments are terminated if the maximum fitness does not change for 50 generations. It is unlikely to have further improvement after that point, so it is assumed that the system converged to a solution. All of the performed experiments were terminated between the 150^{th} and 400^{th} generations and only a few of them are terminated after 300^{th} generation. However, the upper limit is set to a much higher value than 300 not to prevent further improvement. The simulator has to be deterministic in order to test using aforementioned method whether the solution converges to its optimum value or not. The simulator adds noise on the sensory data by default to be realistic, in this case on the GPS signals. The resolution of the GPS is set to its maximum value to make the simulation deterministic (by setting the resolution to 0.0 in the world file).

The optimization framework distributes the simulations to 40 computers. An experiment takes three to five hours depending on the complexity of the creatures evolved and convergence time.

Two different sets of experiments are conducted. In the first set only the Roombots modules are used as the morphological building blocks, in the second set the passive compliant elements are added to the building blocks. Both types of the experiments use the same controller architecture. Minor modifications are done on the parameters. The experiments are explained in detail and the results are discussed in the following sections. Before presenting the results, one of the experiments is examined as a case study in order to explain the characteristics of the evolutionary algorithm. Complementary videos of the evolved robots will be available on the project's page of the BioRob website (`http://biorob.epfl.ch/`).

4.2 An Example Run

This experiment uses only the Roombots as the building blocks. The experiment is terminated at 153^{rd} generation. The robot who has the highest fitness value is composed of 3 modules connected in series. This run reflects common properties of the experiments.

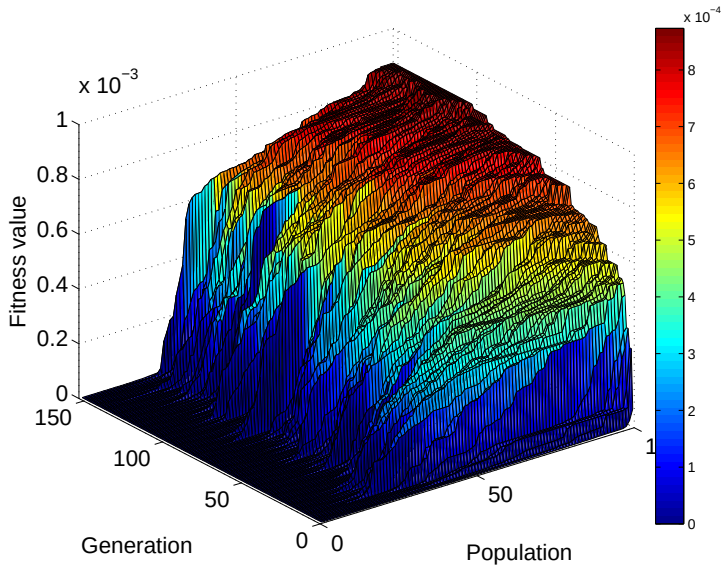


Figure 4.1: Plot of population fitness values along the generations. The fitness values are sorted in the generation. In each generation there are around 10 robots with 0 fitness value. These are self colliding robots. This experiment terminated at 153^{rd} generation. The maximum fitness value increases very fast in first 50 generation. Then, minor changes occur till it reaches its maximum value at 103^{rd} generation.

Figure 4.1 shows the fitness values for the experiment. In this experiment, the system converges to the solution found at 103^{rd} generation. The fitness values are sorted in the generations. All of the experiments have similar fitness plots. The optimization algorithm ensures that there will not be two identical grammars in the same generation. However, there are many individuals with the exact same fitness value. The differences between these grammars does not affect the phenotype, hence they represent same robots. As mentioned before, there are some parts in the genome, non-coding, which does not have an effect on the phenotype. The mutations on these parts does not change the phenotype but create a new valid individual.

Such copies of the fittest individual start to dominate the population.

After all, they have the highest fitness value. Evolutionary operations on these copies help to search fitter solutions around that individual. The system searches the optimum solution around this one in many directions and converges to its optimum.

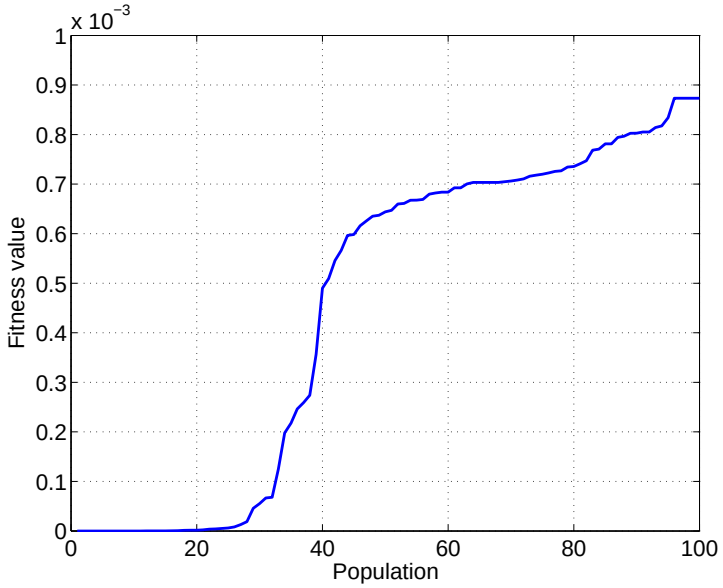


Figure 4.2: Fitness values of the last generation. There are four robots with same phenotype who has the highest fitness value. These have the same phenotype. The population has 14 robots with 0 fitness value.

In Figure 4.2, the fitness values of the last generation are shown. There are 4 individuals with the highest fitness value. This number might be much higher depending on the size of the non-coding regions of the genotype. The other robots who has high fitness values in this population are variants of the fittest one. There are minor changes between them.

As shown in Figure 4.1, there are always individuals with zero fitness in every generation. These are self-colliding robots. The evolution operations may create such individuals. Moreover, randomly replaced individuals may be self-colliding.

The evolutionary algorithm searches for better solutions by evolution operations. In Figure 4.3 , the best fitness values between 20^{th} generation and 50^{th} generation and corresponding impact, speed and torque values are given. Full versions of the graphs are given in Appendix C. The fitness value changed at seven points between these generations. The reasons of the changes are analyzed here. In 23^{rd} generation fitness value increased from 6.1328×10^{-4} to 6.3425×10^{-4} via mutation. The mutation changed

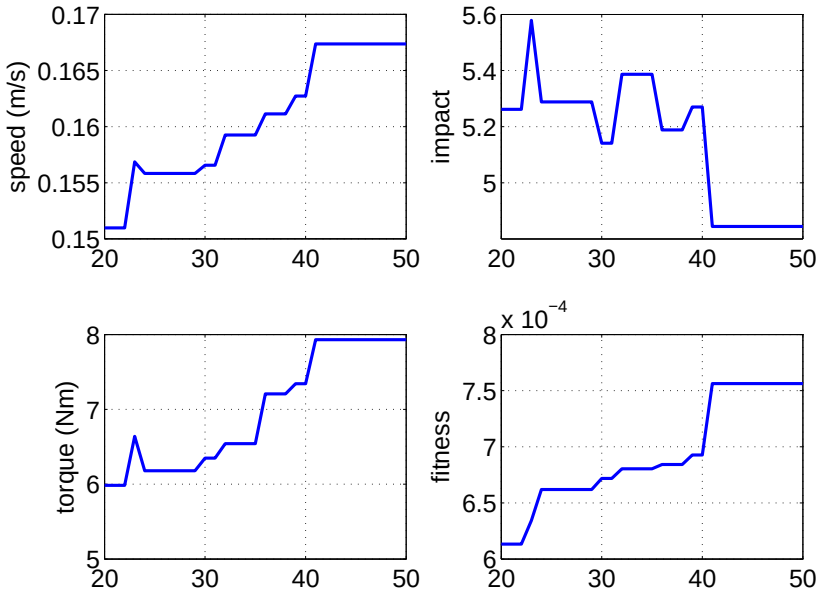


Figure 4.3: Fitness, impact, speed and torque values belong to the best individuals between 20^{th} and 50^{th} are shown in the plots. The values are changed at 7 points.

a parameter in one of the build commands of the previous generation's best individual. In 24^{th} generation a crossover increased the best fitness value. Both of the parent individuals had average fitness values in the population. They had one rule in common but they encoded different morphologies and controllers. In 30^{th} generation best individual is created with crossover and mutation. The parents had exactly same morphologies (world files) and it is also same as the best individual emerged from this experiment. However, their controllers were considerably different. Combination of their controllers reduced the impact of the ground and increased the speed. In 32^{th} generation, a mutation caused the increment. The mutation deleted one of the build commands which changed an offset value. Therefore, it changed an oscillator's offset value .This robot had a very close fitness value to the best one in the previous generation. In 33^{th} and 38^{th} generations a mutation changed an amplitude of the population's fittest robot via changing parameter of the "CHANGE_AMPLITUDE" command. In

40^{th} generation again an amplitude modification incremented the fitness. A "CHANGE_AMPLITUDE" command added to the genome via mutation, also crossover occurred on this genome. But it did not have an effect since only one rule of the parents were different.

In Figure 4.4, the robot emerged from this experiment is shown. The oscillation of the $m1$ servo of the central servo drives the robot.

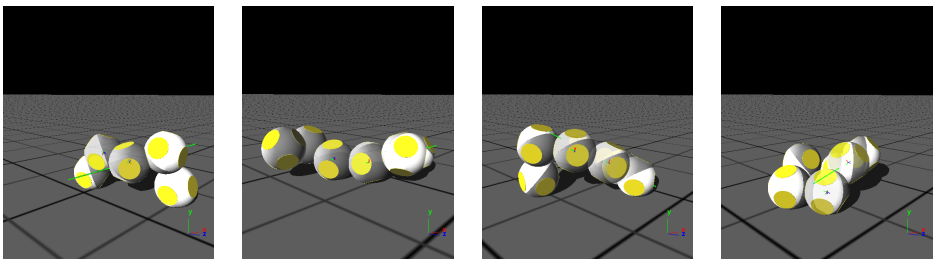


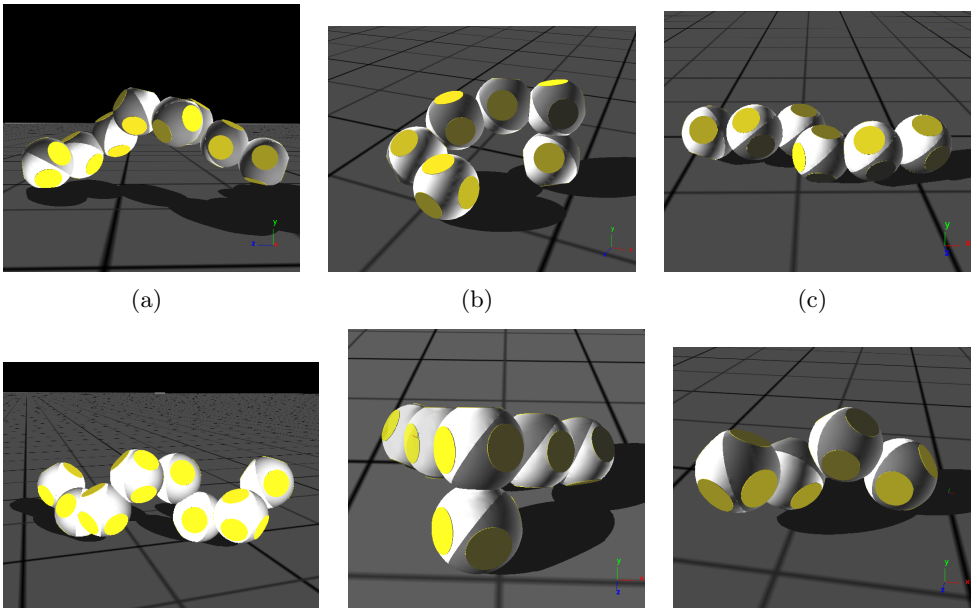
Figure 4.4: Snapshots for the evolved robot, half a cycle is shown. In the first and third images the robot stands on one of the side modules. The robot then pushes itself and puts central module on the floor. The speed of the robot is $17.4cm/s$

4.3 Roombots Experiments

In these experiments only the Roombots modules are used as the morphological building blocks. Many experiments are performed to test the system and to tune the parameters, such as evolutionary algorithm's probabilities and step values of the variables. The step values given in Table 2.2 and parameters explained in Evolutionary Algorithm section are used to perform 16 experiments.

4.3.1 Evolved Gaits

A variety of locomotion strategies are evolved. In Figure 4.5, some of the evolved creatures are shown. The robot (4.5(b)) has a very similar gait to the one shown in Figure 4.4. However, the controller and the morphology of these robots are different. Similar gaits are emerged from four experiments out of 16 with very close speeds. This robot uses the modules at the edges like arms. The oscillations of the $s1$ and $s2$ servo of the centered module rotate the other modules. The robot stands on the modules at the edges then it falls forward. The experiments show that the Roombots robots are able to push themselves and roll in many different ways. It is faster to find such gaits; therefore, the system finds robots which move in these ways more frequently. As shown in Figure 4.5 other types of gaits are also observed.



(a) (b) (c)

(d) (e) (f)

Figure 4.5: Six evolved robots. (a) An evolved robot is shown while it is standing on the modules at the ends, then the robot falls forward and puts rightmost two modules in front of itself. While these modules slide through the right side, the robot stands again. Its speed is $20.4cm/s$. (b) The gait of this robot is like crawling. It uses the modules at the edges to crawl and the front module alternates between left and right modules($18.4cm/s$) . (c) The modules around the centered one moves in the same way (moves in front of centered one) with a phase bias and the robot rolls ($15.3cm/s$). (d) Similar to the robot shown in Figure 4.6, a wave undulates through the body of the robot($33.3cm/s$). (e) The lower spheres of the centered module and the leftmost module rotate and they work like wheels. Leftmost module always keep contact with the floor and the others move smoothly. Therefore, the robot receives low impact from the ground ($11.5cm/s$). (f) After the robot takes the posture shown, it rolls over itself ($11.3cm/s$).

Undulation like gaits are emerged in different experiment runs using four to six modules. In these cases, the robot's shape is snake like and most of the servo motors are driven by oscillatory functions. The phase biases between oscillators create the undulation. An example to such gait is given in Figure 4.6. The genome of this creature encodes the properties of a module and it is replicated four times in phenotype; hence, it benefits from developmental encoding's support for self-similar structures.

The results are not promising by means of symmetry. While, some of the evolved robots' gaits seem as if they were symmetrical, a closer look to

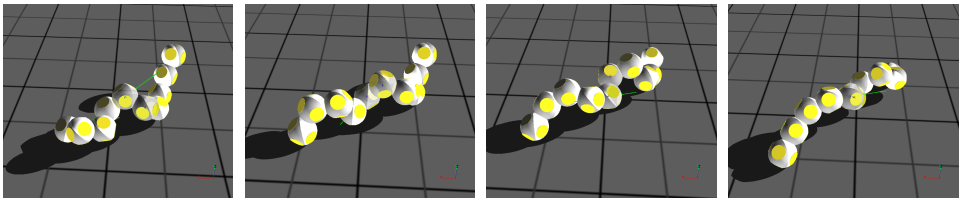


Figure 4.6: Snapshots of an evolved robot, quarter of a cycle is shown. A wave undulates through the body of the robot. This robot has the fastest gait. ($41.2cm/s$)

the controllers shows that they are not actually. The robots shown in Figure 4.5(b) and 4.5(c) exemplifies this situation.

4.3.2 Evaluation of the Results

The algorithm tests robots with two to eight modules, while the robots with three modules dominate the results. In nine experiment runs out of 16, robots with three modules emerged as the fittest robot. As shown in Figure 4.7, one robot evolved with two and seven modules, three with four modules and two with five modules.

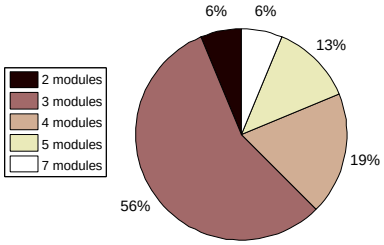


Figure 4.7: Distribution of the results according to the number of modules of the fittest robot.

The average speeds and fitness values according to number of modules which compose the robot are shown in Figure 4.8. The increment in the number of modules also increases the fitness and speed up to four modules. However, it is impractical to conclude with limited data. Even though the robots with four modules have the highest fitness values and the highest speeds, more than 50% of the experiments converged to a robot with three modules. The search space for robots with three modules is narrow than the one for four robots. Moreover, as shown in Figure 4.9 25% of the new created robots have three modules, whereas 15% has four modules. Therefore, a narrower search space is represented with higher probability. Consequently, the evolution converges to a solution with less modules more frequently though

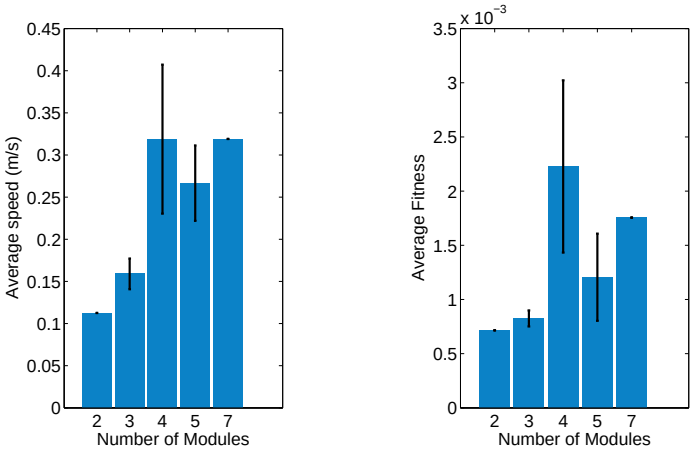


Figure 4.8: These results are grouped by the number of the modules of the robot. The standard deviation data is not available for the last and first bar, because one experiment is used for each of them. (a) The speed values of the fittest creature of each run are averaged according to number of modules of the robot. (b) Similarly, the fitness values are averaged.

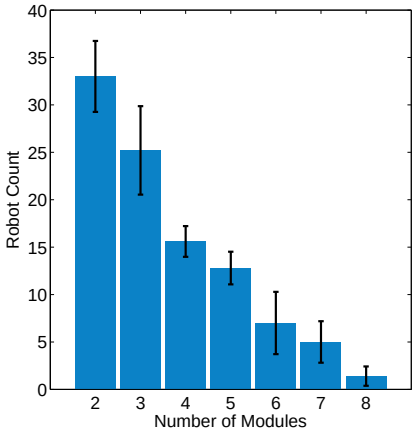


Figure 4.9: The robot counts according to their modules in the initial population. Each generation is composed of 100 individuals.

higher number of modules would support faster gaits. As shown in Figure 4.9, the probability of generation decreases with the increase of the number of modules. The bias towards to smaller robots is needed since their competitors are too strong for them. However, the current system looks like it is too biased since the solution distribution is not uniform. The low probabilities (13%) of creating robots with more than five modules and the lack of large structures in the experimental results justify the previous statement. Though, in every run robots with two modules dominate the initial population, the system converged to such a solution only once in the presented

set of experiments. This result might be rooted from the fact that locomotive ability of a metamodule is limited compared to the others. Spröwitz et al. (2010) used a modified version of the Particle Swarm Optimization algorithm to optimize the speed of the metamodules; the resulted speed of the fastest metamodule is around $17cm/s$. This value gives the upper limit for a metamodule's speed. Since it is the optimal value for the speed, the speeds in the first generations are much less than it. Consequently, the metamodules are eliminated because of low speed values.

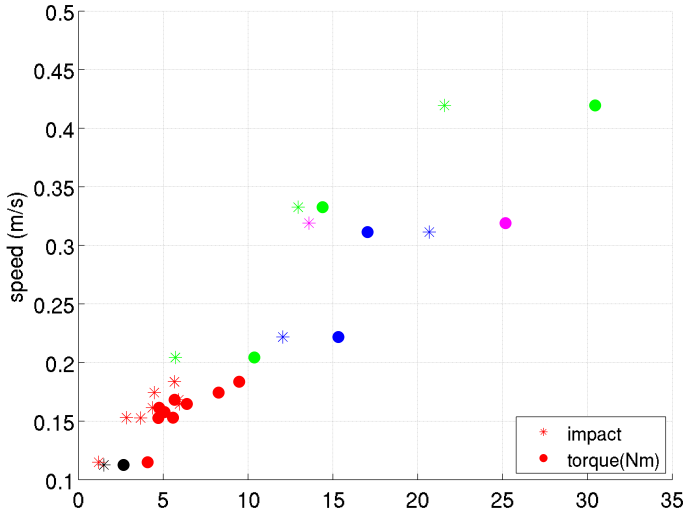


Figure 4.10: Speed vs Impact&Torque. The colors are assigned according to number of modules of the robot. The following scheme is used: black for two modules, red for three modules, green for four modules, blue for five modules and pink for seven modules.

As a final remark, this experiment set shows that the faster the robot is the higher the torque and the impact are (Figure 4.10).

4.4 Roombots and Passive Compliant Elements Experiments

Passive Compliant elements are joints (1 DOF) actuated by a spring-damper system and the spring-damper system is defined by the length of the spring, and spring and damper coefficients.Throughout this research work two different types of placement of passive compliant elements are experienced. Firstly, the elements are attached to only one module and then inserted between the modules. The first type of placement has working principle such as a passive leg. The connection mechanisms of the passive elements

to the modules are the same as the modules' to each other in each type of placement.

4.4.1 Passive Legs

A passive leg changes its length by compression and extension which is known as telescopic motion. The length of the passive legs at its resting state and their positions are evolved. The coefficients for the control mechanism of the passive leg could not be evolved because of the reasons explained in detail in the next section.

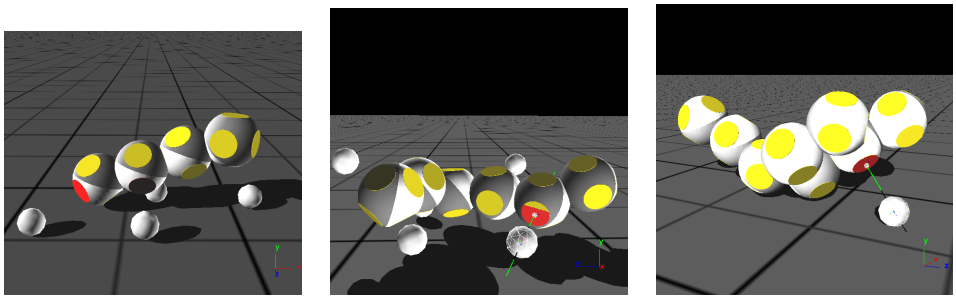
The aim of the experiments is to test the effects of the passive elements on the fitness criteria. Therefore, the optimization parameters are tuned to emphasize on the passive legs. Firstly, the validity criteria for a robot has been changed, previously the robots who has two to eight modules are counted as valid. In these experiments the robots who has two to four modules and two to six passive legs are counted as valid. Consequently, the optimizer forced to use the passive legs. Furthermore, all of the Roombots faces are set as active. Thus, the number of connection faces are set to ten instead of four. The aim is to increase the probability of creation of functioning robots by decreasing probability of self collision which may occur because two elements are tried to be connected to the same connection face. Moreover, an active connection mechanism is not needed for passive elements since they will not be able to detach themselves. Additionally, the population size is set to 200. It is observed that even though the number of active faces are increased, the probability of self collision is increased with the use of passive legs. In order to increase the number of individuals who has more than zero fitness in the first generations, the population size is increased. In the first set of experiments it is observed that, evolution optimizes one of the robots found at first generations. So, it is aimed to increase the variety in the experiment.

16 experiments are performed with the aforementioned properties. The results of these experiments are analyzed in the following sections.

Evolved gaits

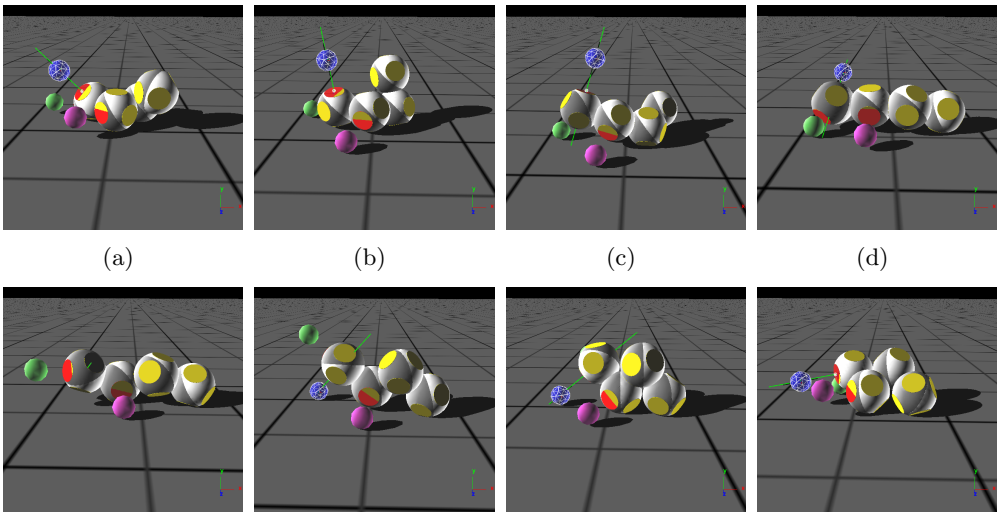
It is observed that, the evolved robots move mostly with two different types of locomotion strategies. The first strategy is rolling by using the energy stored in the spring which enables them to move faster. The second way of locomotion is moving on the passive legs by decreasing the impact of ground. Three of the evolved robots with two, three and four modules are shown in Figure 4.11.

The gait of an evolved robot is explained in Figure 4.12. The robot rolls using energy of the compressed passive leg and rotation of the module at



(a) (b) (c)

Figure 4.11: Three evolved robots. (a) The robot is shown while it is standing on the passive legs. Then, it rolls and makes three compete rounds and stands again on the legs. The speed of the robot is $32.1cm/s$. (b) The robot moves on the passive legs and the modules rarely touch to the ground ($30.7cm/s$). (c) A robot with four modules and two passive legs is shown. One of the legs is behind the modules. The rightmost five module spheres oscillates side to side ($27.8cm/s$).



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 4.12: Snapshots of an evolved robot, a cycle is shown. The rotation of the module at right hand side is clear in (a) (b) (c) (d). (c) The green passive leg starts to get compressed and the pink leg supports the posture. (d) The green leg is compressed as much as possible, then the robot rolls. As shown in (e), (f) and (g) the blue and pink legs support the robot and the module at the left side does not hit to the ground. Then, the green leg touches to the ground again. The speed of the robot is $14.9cm/s$.

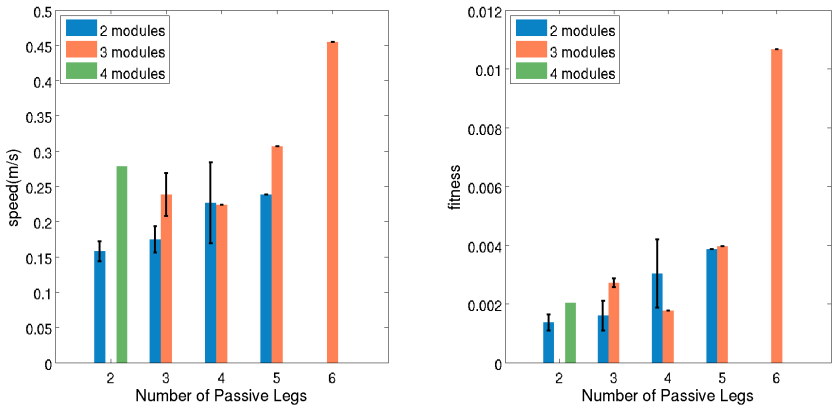
	2l	**3l**	**4l**	**5l**	**6l**
2m	2	4	3	1	0
3m	0	2	0	1	1
4m	1	0	0	0	0

Table 4.1: Distribution of the evolved robots. The fields in the first row contain the number of passive legs; the fields in the first column contain the number of modules. The table shows how many robots are evolved with the given number of passive legs and modules.

the right hand side. Moreover, the legs reduces contacts with the ground of the module on the left.

Evaluation of the Results

The algorithm tests robots composed of two to four modules and two to six passive legs. Only one of the experiments resulted in a robot with four modules and this robot has two passive legs. Robots with two modules dominate the results; 10 experiments out of 16 resulted in robots composed of two modules. Two of these robots have two passive legs, four of them have three passive legs and one of them has one passive leg. Table 4.1 gives the summary of evolved robots according to their passive legs and module counts.



(a) (b)

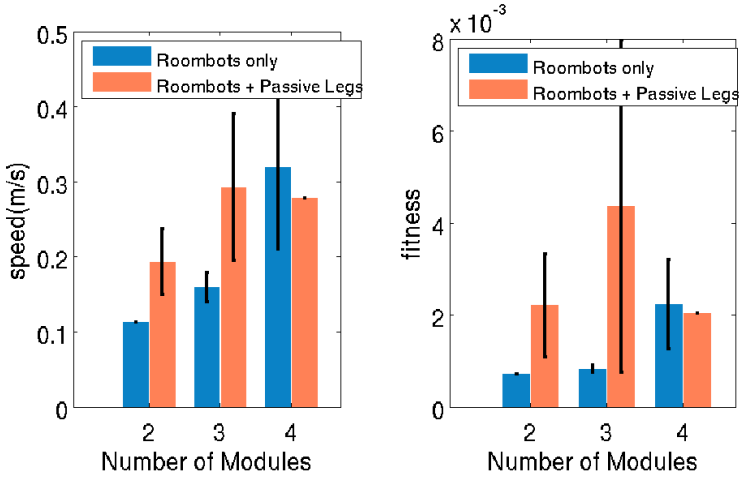
Figure 4.13: The average speeds and fitness values according to number of modules and passive legs are shown.

In Figure 4.13 average speed and fitness values of the fittest robots are shown. The values are averaged according to the number of modules and passive legs employed. In general the fitness and the speed increases with the

increment of the number of passive legs or the number of modules. However, it is not the case for the robot with four modules. Though, it is not realistic to have implication from such a limited data. The detailed results are given in Appendix D as a table.

Comparison of the Results

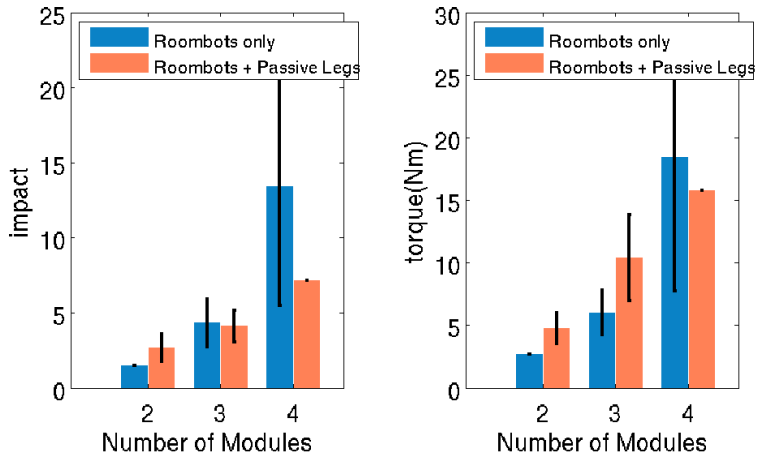
The results of the two experiment sets are compared based on the number of modules which compose the robots. The average speeds and the fitness values are shown in Figure 4.14. The Figure 4.14 shows that use of passive elements increases the speed of the robots with two and three modules. In two modules case, the robots with passive legs have speeds between $0.15m/s$ and $0.29m/s$ which is beyond the speed of a metamodule. The fitness values of the experiments with passive legs are also much higher than the experiments with only the Roombots.



(a) (b)

Figure 4.14: The average speeds and fitness values according to number of modules of the robots.

The average impact and torque values are given in Figure 4.15. Since the passive legs reduce contact of the modules with the ground, it was expected to have lower impact values with the use of passive legs. However, the experiments showed the opposite. The robots move faster with passive legs and they hit to the ground stronger. Therefore, the impact values are increased even though the modules touch the ground much rare. The torque values are also increased and the change in the torque values are parallel with the change of the speeds.



(a) (b)

Figure 4.15: The average impact and torque values according to number of modules of the robots.

4.4.2 Passive Elements Between Modules

The passive compliant elements inserted between modules can move in two ways: rotationally or linearly. The length and the coefficient of the spring are defined as the optimization parameters while the damper coefficient is set to 1% of the spring coefficient. Simulations expose unstable behavior of the passive elements, and undesired results such as explosion of the robot, disconnection of the modules and unrealistic movements are observed. These results might arise from randomly initialized parameters.

The problem might be solved by finding a proper parameter set and fixing the parameters of the passive elements. However, it is almost impossible to find such a parameter set. The passive element which is working as expected with in a robot may cause another one to explode. Therefore, they are set as optimization parameters. The reason of this faulty behaviors could not be found so it is required to detect the robots which moves unrealistically. It is easy to detect broken or disconnected robots by using positions and zero fitness is assigned to such creatures.

The robots spinning around very fast cause some problems. Since their speed can reach up to $2m/s$, which is a very high speed, the selection mechanism works in favor of them. To handle this problem, a gyroscope which is angular velocity measurement device is employed. A threshold is set to eliminate the erroneous robots. However, the thresholding does not solve the problem definitely. A faulty robot is evolved with lower gyroscope measurements than a robot without passive elements. The experiments with these passive elements could not be performed because of the faulty robots.

The modelling of the modules and the passive elements might cause these anomalies. All of them are implemented as individual robots and moved together via connection mechanisms in simulation environment. These are very stiff joints and the physics might cause problems using these mechanisms and spring-damper systems. Further analysis required to spot the reasons of the erroneous movements.

Chapter 5 Conclusion

In many projects controllers for a limited number of hand designed modular robot structures are optimized in order to achieve a given task. However, modular robotics offers a variety of options to construct large structures. In this thesis, a method inspired from biological developmental process is proposed to evolve the morphology and the controllers of the modular robots to accomplish a given task.

The build process of the Roombots robots are encoded using L-systems. Then, an evolutionary algorithm is designed to evolve the L-systems. A builder which decodes the L-systems and prepares the robot files for simulation is developed. The encoded robots are built and evaluated in a physics based simulator. The passive telescopic legs are integrated to the system.

The performed experiments show that robots constructed by the Roombots are able to move in many different ways. Moreover, the experiments used the Roombots and passive telescopic legs as morphological building blocks present that the robots are able to move faster by exploiting the energy stored in the passive legs.

In this work, safe and energy efficient locomotion is employed as the fitness criteria. However, the developed system does not have a limitation on the fitness criteria. Therefore, the robots can be co-evolved for any tasks, it is sufficient to modify the simulator controller to calculate any desired fitness function.

The developed system is open for further developments, because of the limited time only the telescopic passive legs are tested as building blocks in addition to the Roombots. However, any other passive elements can be added with minor modifications. Consequently, while developing passive elements for the Roombots, the system can be used to test them and to tune their parameters.

Future Work

As the future work, sensors such as distance might be added to the robots. More complex tasks such as following an object can be set as evolution task with sensory feedback. The sensor can be treated as one of the building blocks and with an extra build command it can be well integrated to the

L-system. The controller already supports the integration of the sensory feedback (Spröwitz et al., 2010). As mentioned in the experiments chapter, the random generation of the robots is biased towards the small robots. Additional attention required at this point while creating the initial population. Moreover, a stable modelling for passive elements between modules is needed as indicated the experiments chapter. In the current case the torque values are measured to mimic energy consumption which might be modelled in a more accurate way.

References

Biemont, C., & Vieira, C. (2006, October 04). Genetics: Junk dna as an evolutionary force. *Nature*, *443*(7111), 521–524.

Cpg-network reference manual. (2010). `http://biorob2.epfl.ch/users/jvanden/docs/cpg-network/`.

Endo, K., Maeno, T., & Kitano, H. (2002). Co-evolution of morphology and walking pattern of biped humanoid robot using evolutionary computation. consideration of characteristic of the servomotors. , *3*, 2678 - 2683 vol.3.

Floreano, D., & Mattiussi, C. (2008). *Bio-inspired artificial intelligence.* New York: McGraw-Hill.

Hornby, G. S., Lipson, H., & Pollack, J. B. (2001). Evolution of generative design systems for modular physical robots. In *In ieee international conference on robotics and automation* (pp. 4146–4151).

Hornby, G. S., & Pollack, J. B. (2001a). The advantages of generative grammatical encodings for physical design. In *In congress on evolutionary computation* (pp. 600–607). IEEE Press.

Hornby, G. S., & Pollack, J. B. (2001b, 7-11 July). Body-brain co-evolution using L-systems as a generative encoding. In L. Spector et al. (Eds.), *Proceedings of the genetic and evolutionary computation conference (gecco-2001)* (pp. 868–875). San Francisco, California, USA: Morgan Kaufmann.

Ijspeert, A., Crespi, A., Ryczko, D., & Cabelguen, J.-M. (2007). From swimming to walking with a salamander robot driven by a spinal cord model [article]. *Science*, *315*(5817), 1416–1420. Available from `http://dx.doi.org/10.1126/science.1138353` (Additional material, like the supporting online material, movies and pictures can be accessed via the .html link.)

Ijspeert, A. J. (2008). Central pattern generators for locomotion control in animals and robots: a review [article]. *Neural Networks*, *21*(4), 642–653.

Komosinski, M. (2000). The world of framsticks: Simulation evolution interaction. In *In: Proceedings of 2 nd international conference on virtual worlds* (pp. 214–224). Springer-Verlag.

Komosinski, M., & Rotaru-Varga, A. (2002). Comparison of different geno-

type encodings for simulated three-dimensional agents. *Artif. Life*, *7*(4), 395–418.

Kurokawa, H. K., Yoshida, A., Tomita, E., Kokaji, K., & S. Murata, S. (2003, oct). M-tran ii: metamorphosis from a four-legged walker to a caterpillar. , 2454–2459.

liboptimization-sharp api reference. (2010). `http://biorob2.epfl.ch/users/jvanden/docs/optimization-sharp/`. ([Online; accessed 5-June-2010])

Lipson, H., & Pollack, J. B. (2000, August 31). Automatic design and manufacture of robotic lifeforms. *Nature*, *406*(6799), 974–978. Available from `http://dx.doi.org/10.1038/35023115`

Marbach, D., & Ijspeert, A. J. (2004). Co-evolution of configuration and control for homogenous modular robots. In *Proceedings of the eighth conference on intelligent autonomous systems (ias8* (pp. 712–719). IOS Press.

Mayer, M. (2009). *Roombot modules - kinematics considerations for moving optimizations.* Available from `http://birg.epfl.ch/page69834.html`

Michel, O. (2004). Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, *1*(1), 39–42. Available from `http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-1/39-42.pdf`

Optimization framework conceptual overview. (2010). `http://biorob2.epfl.ch/users/jvanden/docs/optimization-concept/concept.pdf`. ([Online; accessed 5-June-2010])

Pollack, J. B., Lipson, H., Hornby, G., & Funes, P. (2001). Three generations of automatically designed robots. *Artificial Life*, *7*, 2001.

Prusinkiewicz, P., & Lindenmayer, A. (1990). *The algorithmic beauty of plants.* New York: McGraw-Hill.

Righetti, L., & Ijspeert, A. J. (2006). Programmable Central Pattern Generators: an application to biped locomotion control. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation.*

Salemi, B., Moll, M., & Shen, W.-M. (2006, 9-15). Superbot: A deployable, multi-functional, and modular self-reconfigurable robotic system. In *Intelligent robots and systems, 2006 ieee/rsj international conference on* (p. 3636 -3641).

Sims, K. (1994a). Evolving 3d morphology and behavior by competition. *Artif. Life*, *1*(4), 353–372.

Sims, K. (1994b). Evolving virtual creatures. In *Siggraph '94: Proceedings of the 21st annual conference on computer graphics and interactive techniques* (pp. 15–22). New York, NY, USA: ACM.

Sproewitz, A., Billard, A., Dillenbourg, P., & Ijspeert, A. J. (2009). Roombots-Mechanical Design of Self-Reconfiguring Modular Robots

for Adaptive Furniture. In *Proceedings of 2009 IEEE International Conference on Robotics and Automation* (pp. 4259–4264).

Sproewitz, A., Moeckel, R., Maye, J., Asadpour, M., & Ijspeert, A. J. (2007). Adaptive Locomotion Control in Modular Robotics. In *Workshop on Self-Reconfigurable Robots/Systems and Applications IROS07* (pp. 81–84).

Spröwitz, A., Pouya, S., Bonardi, S., van den Kieboom, J., Möckel, R., Billard, A., et al. (2010). Roombots: Reconfigurable Robots for Adaptive Furniture. *IEEE Computational Intelligence Magazine, special issue on "Evolutionary and developmental approaches to robotics"*.

Webots. (n.d.). *http://www.cyberbotics.com.* Available from `http://www.cyberbotics.com` (Commercial Mobile Robot Simulation Software)

Appendix A The CPG-network File

```
<?xml version="1.0" encoding="UTF-8"?>
<cpg>
  <network>
  <globals>
    <property name="frequency">0.25</property>
  </globals>

    <templates>
      <state id="oscillator">
        <property name="R">1</property>
        <property name="P">0</property>
        <property name="X">0</property>
        <property name="x" integrated="yes">0</property>
        <property name="p" integrated="yes">x</property>
        <property name="r" integrated="yes">0.0</property>
        <property name="drive">r * sin(p) + x</property>
      </state>
      <link id="integrate">
        <property name="b">2</property>
  <action target="x"> b*( X - x)</action>
        <property name="a">2</property>
        <action target="r"> a * (R - r ) </action>
        <action target="p">frequency * 2 * PI</action>
      </link>
      <link id="coupling">
        <property name="phase">0</property>
        <property name="weight">2</property>
        <action target="p">weight *from.r* sin(from.p - to.p - phase)</action>
      </link>
    </templates>


  <!-- Module 1 -->
  <state id="module_1::s1" ref="oscillator">
    <property name="drive">r*sin(p)+x</property>
    <property name="R">1.5</property>
    <property name="X">0</property>
  </state>
  <state id="module_1::m1" ref="oscillator">
    <property name="drive">-p</property>
    <property name="R">1.5</property>
    <property name="X">0</property>
  </state>
  <state id="module_1::s2" ref="oscillator">
    <property name="drive">r*sin(p)+x</property>
    <property name="R">1.5</property>
    <property name="X">-0.349066</property>
  </state>
  <link id="int_m1_s1" ref="integrate" from="module_1::s1" to="module_1::s1"/>
  <link id="int_m1_m1" ref="integrate" from="module_1::m1" to="module_1::m1"/>
  <link id="int_m1_s2" ref="integrate" from="module_1::s2" to="module_1::s2"/>
  <!-- Inter module 1 coupling -->
  <link id="intmod_m1_s1_m1" ref="coupling" from="module_1::s1" to="module_1::m1"
      >
     <property name="phase">0</property>
  </link>
  <link id="intmod_m1_m1_s1" ref="coupling" from="module_1::m1" to="module_1::s1"
      >
    <property name="phase">-0</property>
  </link>
  <link id="intmod_m1_m1_s2" ref="coupling" from="module_1::m1" to="module_1::s2"
      >
```

```
61      <property name="phase">0</property>
    </link>
63  <link id="intmod_m1_s2_m1" ref="coupling" from="module_1::s2" to="module_1::m1"
        >
      <property name="phase">-0</property>
65  </link>
    <!-- Module 2 -->
67  <state id="module_2::s1" ref="oscillator">
      <property name="drive">x</property>
69    <property name="R">1.5</property>
      <property name="X">0</property>
71  </state>
    <state id="module_2::m1" ref="oscillator">
73    <property name="drive">r*sin(p)+x</property>
      <property name="R">2.9</property>
75    <property name="X">-1.22173</property>
    </state>
77  <state id="module_2::s2" ref="oscillator">
      <property name="drive">x</property>
79    <property name="R">2.9</property>
      <property name="X">2.61799</property>
81  </state>
    <link id="int_m2_s1" ref="integrate" from="module_2::s1" to="module_2::s1"/>
83  <link id="int_m2_m1" ref="integrate" from="module_2::m1" to="module_2::m1"/>
    <link id="int_m2_s2" ref="integrate" from="module_2::s2" to="module_2::s2"/>
85  <!-- Inter module 2 coupling -->
    <link id="intmod_m2_s1_m1" ref="coupling" from="module_2::s1" to="module_2::m1"
        >
87    <property name="phase">1.5708</property>
    </link>
89  <link id="intmod_m2_m1_s1" ref="coupling" from="module_2::m1" to="module_2::s1"
        >
      <property name="phase">-1.5708</property>
91  </link>
    <link id="intmod_m2_m1_s2" ref="coupling" from="module_2::m1" to="module_2::s2"
        >
93    <property name="phase">-2.44346</property>
    </link>
95  <link id="intmod_m2_s2_m1" ref="coupling" from="module_2::s2" to="module_2::m1"
        >
      <property name="phase">--2.44346</property>
97  </link>
    <!-- Module 3 -->
99  <state id="module_3::s1" ref="oscillator">
      <property name="drive">x</property>
101   <property name="R">1.5</property>
      <property name="X">0</property>
103 </state>
    <state id="module_3::m1" ref="oscillator">
105   <property name="drive">r*sin(p)+x</property>
      <property name="R">2.9</property>
107   <property name="X">-1.22173</property>
    </state>
109 <state id="module_3::s2" ref="oscillator">
      <property name="drive">x</property>
111   <property name="R">2.9</property>
      <property name="X">2.61799</property>
113 </state>
    <link id="int_m3_s1" ref="integrate" from="module_3::s1" to="module_3::s1"/>
115 <link id="int_m3_m1" ref="integrate" from="module_3::m1" to="module_3::m1"/>
    <link id="int_m3_s2" ref="integrate" from="module_3::s2" to="module_3::s2"/>
117 <!-- Inter module 3 coupling -->
    <link id="intmod_m3_s1_m1" ref="coupling" from="module_3::s1" to="module_3::m1"
        >
119   <property name="phase">-2.96706</property>
    </link>
121 <link id="intmod_m3_m1_s1" ref="coupling" from="module_3::m1" to="module_3::s1"
        >
      <property name="phase">--2.96706</property>
123 </link>
    <link id="intmod_m3_m1_s2" ref="coupling" from="module_3::m1" to="module_3::s2"
        >
125   <property name="phase">-2.44346</property>
    </link>
127 <link id="intmod_m3_s2_m1" ref="coupling" from="module_3::s2" to="module_3::m1"
        >
      <property name="phase">--2.44346</property>
129 </link>
    <!-- Module 4 -->
131 <state id="module_4::s1" ref="oscillator">
      <property name="drive">x</property>
133   <property name="R">1.5</property>
      <property name="X">0</property>
135 </state>
```

```
    <state id="module_4::m1" ref="oscillator">
      <property name="drive">r*sin(p)+x</property>
      <property name="R">2.6</property>
      <property name="X">-1.22173</property>
    </state>
    <state id="module_4::s2" ref="oscillator">
      <property name="drive">x</property>
      <property name="R">2.9</property>
      <property name="X">2.61799</property>
    </state>
    <link id="int_m4_s1" ref="integrate" from="module_4::s1" to="module_4::s1"/>
    <link id="int_m4_m1" ref="integrate" from="module_4::m1" to="module_4::m1"/>
    <link id="int_m4_s2" ref="integrate" from="module_4::s2" to="module_4::s2"/>
    <!-- Inter module 4 coupling -->
    <link id="intmod_m4_s1_m1" ref="coupling" from="module_4::s1" to="module_4::m1"
        >
      <property name="phase">1.5708</property>
    </link>
    <link id="intmod_m4_m1_s1" ref="coupling" from="module_4::m1" to="module_4::s1"
        >
      <property name="phase">-1.5708</property>
    </link>
    <link id="intmod_m4_m1_s2" ref="coupling" from="module_4::m1" to="module_4::s2"
        >
      <property name="phase">-2.44346</property>
    </link>
    <link id="intmod_m4_s2_m1" ref="coupling" from="module_4::s2" to="module_4::m1"
        >
      <property name="phase">--2.44346</property>
    </link>
     <!-- Coupling between modules -->
    <link id="sepmod_m1_m2" ref="coupling" from="module_1::m1" to="module_2::m1">
      <property name="phase"> -1.22173 </property>
    </link>
    <link id="sepmod_m2_m1" ref="coupling" from="module_2::m1" to="module_1::m1">
      <property name="phase"> --1.22173 </property>
    </link>
    <link id="sepmod_m2_m3" ref="coupling" from="module_2::m1" to="module_3::m1">
      <property name="phase"> -1.22173 </property>
    </link>
    <link id="sepmod_m3_m2" ref="coupling" from="module_3::m1" to="module_2::m1">
      <property name="phase"> --1.22173 </property>
    </link>
    <link id="sepmod_m3_m4" ref="coupling" from="module_3::m1" to="module_4::m1">
      <property name="phase"> -1.22173 </property>
    </link>
    <link id="sepmod_m4_m3" ref="coupling" from="module_4::m1" to="module_3::m1">
      <property name="phase"> --1.22173 </property>
    </link>
    </network>
</cpg>
```

cpg.xml

Appendix B The Roombots World Description File

```
<roombot_config>
  <module_tree>
    <module id_module="1">
      <global_position>
      <translation x="0.00" y="0.44" z="0.00"/>
      </global_position>
      <module id_module="2">
        <connection connector_parent="C0X" connector_child="C1Y" type="PER"/>
        <module id_module="3">
          <connection connector_parent="C2Y" connector_child="C1Y" type="PER"/>
          <module id_module="4">
            <connection connector_parent="C2Y" connector_child="C1Y" type="PER"/>
          </module>
        </module>
      </module>
    </module>
  </module_tree>
  <parameter>
    <supervisor_controller>
      <name>supervisor</name>
      <arguments> -n cpg.xml -f 0.25 </arguments>
    </supervisor_controller>
    <roombot_controller>
      <name>module</name>
    </roombot_controller>
  </parameter>
</roombot_config>
```

world.xml

Appendix C Experiment Plots

C.1 Fitness Plot

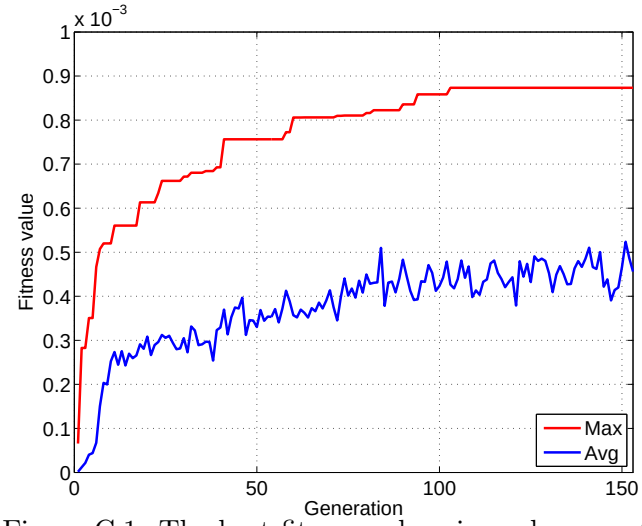


Figure C.1: The best fitness values in each generation

C.2 Impact Plot

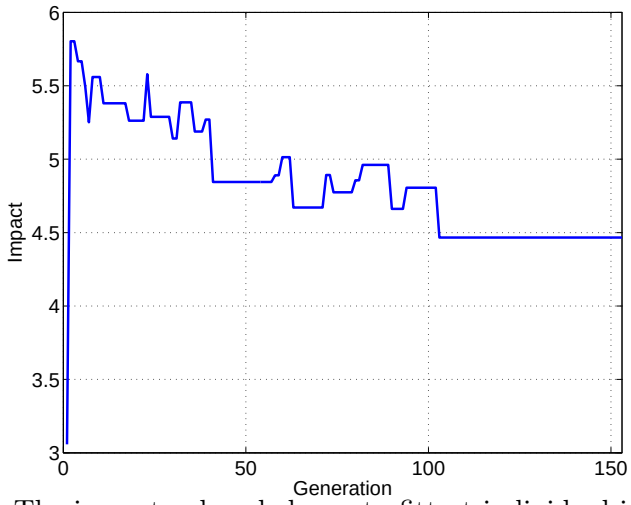


Figure C.2: The impact values belongs to fittest individual in each generation

C.3 Torque Plot

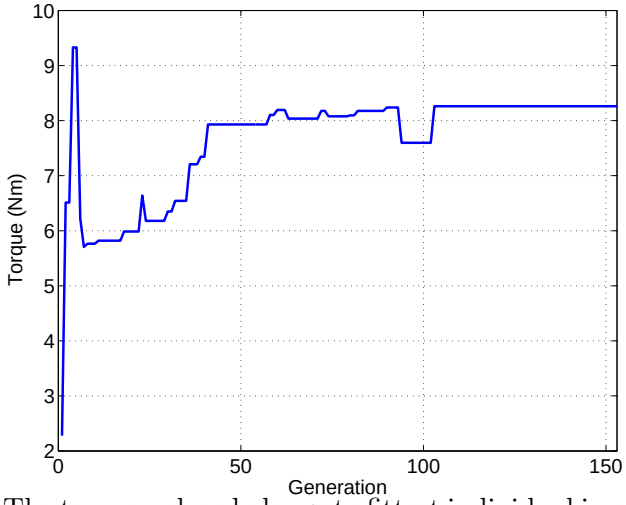


Figure C.3: The torque values belongs to fittest individual in each generation

C.4 Speed Plot

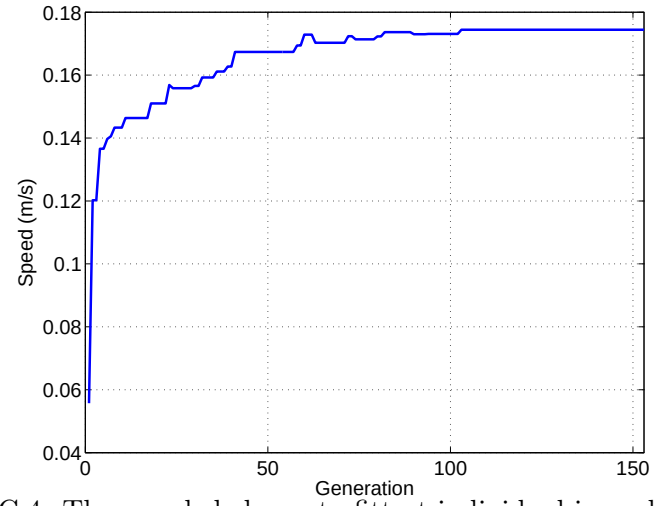


Figure C.4: The speeds belongs to fittest individual in each generation

Appendix D The Results of the Experiments

D.1 Roombots Experiments

#	**Iteration**	**Impact**	**Torque(Nm)**	**Speed(cm/s)**	**Fitness(x 10^{-4})**
2	190	1.4971	2.6603	11.2635	7.1603
3	153	4.4665	8.2612	17.4409	8.7337
3	158	4.8557	5.0756	15.7699	7.8998
3	160	1.1898	4.0783	11.4907	6.8876
3	173	3.6577	4.7019	15.2657	8.5785
3	190	5.6561	9.4743	18.3577	8.4514
3	210	5.9114	5.6599	16.8285	8.2392
3	255	5.9604	6.3919	16.4736	7.2429
3	288	2.8259	5.5676	15.2915	9.0144
3	364	4.3549	4.7512	16.1363	9.2368
4	248	12.9524	14.3861	33.2590	26.9513
4	278	21.5753	30.4592	41.9500	28.7977
4	321	5.7097	10.3626	20.4300	11.0857
5	156	20.6778	17.0418	31.1339	16.0765
5	230	12.0458	15.3144	22.1815	8.0353
7	269	13.5883	25.1703	31.9028	17.5574

Table D.1: Each row gives information regarding an experiment. The first column shows the module count of the fittest robot of the experiment. The second column shows the generation when the experiment is terminated. The rest of the columns give data about the fittest robot.

D.2 Experiments with Passive Legs

#	PL	**Iteration**	**Impact**	**Torque(Nm)**	**Speed(cm/s)**	**Fitness(x 10^{-4})**
2	2	151	1.4688	5.1666	14.7720	11.7015
2	2	157	2.8279	3.2629	16.7904	15.5829
2	3	151	3.8587	4.3630	17.0994	12.1851
2	3	162	2.8432	5.6268	19.6437	18.9511
2	3	196	2.4216	4.0256	15.1710	11.1837
2	3	377	2.3648	3.0763	17.9660	21.5005
2	4	151	1.5849	4.1881	17.8347	22.0182
2	4	165	4.4306	7.0675	29.0135	43.6450
2	4	173	2.4992	5.5472	21.1539	25.4232
2	5	279	2.2498	5.4641	23.8428	38.6583
3	3	164	4.3950	8.7727	25.9713	28.2119
3	3	183	2.3139	6.5991	21.6932	26.1251
3	4	335	4.4509	9.0219	22.3661	17.6562
3	5	297	4.3315	12.3670	30.7000	39.5333
3	6	369	5.0805	15.2670	45.4639	106.7017
4	2	317	7.1300	15.8079	27.8340	20.3116

Table D.2: Each row gives information regarding an experiment. The first column shows the module count of the fittest robot of the experiment. The second column shows the number of the passive legs of the fittest robot. The third column shows the generation when the experiment is terminated. The rest of the columns give data about the fittest robot.