Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Jan Černý

Study programme: Open Informatics

Specialisation: Artificial Intelligence

Title of Diploma Thesis: Evolutionary Design of Robot Motion Patterns

Guidelines:

1. Get acquainted with the robotic simulator SIM, see http: /lynx1.felk.cvut.cz/~danfis/sim-doc/, and learn to use it.
2. Design an evolutionary algorithm for learning effective motion patterns of robot in the SIM simulator.
3. Implement the proposed evolutionary algorithm.
4. Design proof-of-concept experiments and experimentally evaluate the performance of the evolutionary system, the quadruped robot creature being the goal experiment.

Bibliography/Sources: Will be provided by the supervisor.

Diploma Thesis Supervisor: Ing. Jiří Kubalík, Ph.D.

Valid until: the end of the summer semester of academic year 2012/2013

L.S.

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2012

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Jan Černý

Studijní program: Otevřená informatika (magisterský)

Obor: Umělá inteligence

Název tématu: Evoluční návrh strategie řídící pohyb robotu

Pokyny pro vypracování:

1. Seznamte se s robotickým simulátorem SIM, viz http://lynx1.felk.cvut.cz/~danfis/sim-doc/.
2. Navrhněte evoluční algoritmus pro učení efektivních pohybových vzorců robotu s použitím simulátoru SIM.
3. Implementujte navržený evoluční algoritmus.
4. Navrhněte a proveďte základní experimenty pro ověření funkčnosti evolučního algoritmu. Proveďte a vyhodnoťte experimenty se čtyřnohým robotem.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí diplomové práce: Ing. Jiří Kubalík, Ph.D.

Platnost zadání: do konce letního semestru 2012/2013

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2012

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Master's Thesis

Evolutionary Design of Robot Motion Patterns

Bc. Jan Černý

Supervisor: Ing. Jiří Kubalík, Ph.D.

Study Programme: Otevřená Informatika, strukturovaný, Navazující magisterský

Field of Study: Umělá Inteligence

May 13, 2012

Aknowledgements

I would like to thank my supervisor Ing. Jiří Kubalík, Ph.D. whose guidance allowed me to overcome all difficulties and finish this work and to Ing. Martin Dobiáš for his help and moral support.

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 11. 5. 2012 ……

……

Podpis autora práce

Abstract

This thesis is focused on the use and implementation of Genetic Programming for generating viable motion patterns for robotic creatures. SYMBRION and REPLICATOR are two European projects whose research is focused on application of biological knowledge in robotics. One of the robots developed as a part of these projects is used in this work as a building block for two larger four legged robotic organisms.

A co-evolution algorithm has been developed to generate single leg movements and adapt them to the three remaining legs. This approach of dividing the problem into two smaller sub-problems simplifies the evolution and saves the processing time. It is shown that the implemented Evolution Algorithm is indeed capable of generating motion patterns for robots very similar to those seen in nature and that by using them the robots are able to efficiently reach their predefined targets. All the experiments are conducted in a simulated environment.

Abstrakt

Tato práce se zaměřuje na využití a implementaci Genetického Programování pro generování pohybových vzorů pro robotické organismy. SYMBRION a RREPLICATOR jsou dva evropské projekty, jejichž výzkum je zaměřen na použití biologických poznatků v robotice. Jeden z robotů, vyvinutých v rámci těchto projektů, je v této práci využit jako stavební blok pro dva větší čtyřnohé robotické organismy.

V rámci této práce byl vyvinut koevoluční algoritmus, který generuje pohybové vzory pro jednu nohu a tyto upravuje pro zbývající tři nohy. Tento přístup rozdělení problému na dva menší podproblémy zjednodušuje evoluci a tím šetří výpočetní čas. Bylo ukázáno, že implementovaný Evoluční Algoritmus je schopný generovat pohybové vzory velmi podobné těm, které lze pozorovat v přírodě a roboti, kteří je používají, jsou schopní efektivně dosáhnout definovaného cíle. Všechny experimenty byly prováděny v simulovaném prostředí.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

SYMBRION and REPLICATOR are two projects that connect several universities from across the whole Europe. Both projects are focused on robotics and their pivotal objective is to apply knowledge of biology, evolution and swarm behaviour into this rapidly developing field. One of the robots developed as a part of these project on Scuola Superiore Sant'Anna in Italy is used as elemental building block in this work. The goal of this thesis is to devise and implement an algorithm which will be able to generate an efficient motion pattern for a robotic organism constructed from these robots. It is expected that this motion will resemble those that have already evolved and can be observed in nature, as those kinds of movements are considered efficient and elegant. This goal is going to be pursued by the use of Evolution Algorithms.

Evolution Algorithms are a class of optimization methods inspired by the Darwinian theory of evolution and natural selection. They are global search methods that do not use gradient information; they are also robust and very general. This makes them an ideal approach for the black-box type of problems where there is no or very little information about the optimized function.

Genetic Programming is an Evolutionary Algorithm which is based on the same principles of evolution and natural selection, but is instead evolving solutions represented by a string of numbers, it evolves whole computer programs which can be used to solve a whole spectrum of different problems or perform a specific task. One of the problems commonly solved by Genetic programming is function approximation and since the movements of the robots can be easily represented by a function or a set of functions Genetic Programming can be a perfect device to accomplish the goal of this work.

One of the important aspects of this work is the use of a robotic simulator instead of real robots. This allowed evaluation of thousands individuals of the Evolution Algorithm in a fraction of the time which would normally be needed. This speed-up allowed to evolve large populations for many generations which would otherwise be impossible. The use of the robotic simulator instead of physical robots is a critical condition leading to the accomplishment of the set goals.

The work is divided into 8 chapters, with Chapter 1 being this introduction. Chapter 2 contains the most important facts about the used robotic platform and states the main objectives of this work. In Chapter 3 there is an introduction into the basic principles of computer simulations of robots which are among the most important components of all experiments. An introduction into Genetic Algorithms and Genetic Programming which are the focus points of this work and the another pivotal component of the experiments is in Chapter 4. In Chapter 5 the proposed solution of the stated optimization problem is introduced. Chapter 6 then describes all important details of the implementation. Description of all conducted experiments is the first part of Chapter 7. The second part of this chapter contains description and evaluation of all obtained results. Chapter 8 contains the conclusion.

Chapter 2

Problem and Goals Description

2.1 SYMBRION and REPLICATOR Projects

SYMBRION (Symbiotic Evolutionary Robot Organisms) and REPLICATOR (Robotic Evolutionary Self-Programming and Self-Assembling Organisms) are two project focused on investigation and development of multi-robot organism. The main focus of these projects is to investigate and develop novel principles of adaptation and evolution for symbiotic multi-robot organisms based on bio-inspired approaches and modern computing paradigms. Such robot organisms consist of swarms of robots, which can dock with each other and symbiotically share energy and computational resources within a single artificial-life-form. When it is advantageous to do so, these swarm robots can dynamically aggregate into one or many symbiotic organisms and collectively interact with the physical world via a variety of sensors and actuators [8]. This leads not only to extremely adaptive, evolve-able and scalable robotic systems, but also enables robot organisms to reprogram themselves without human supervision and for new, previously unforeseen, functionality to emerge [9].

2.2 Description of Single Robotic Organisms

Robots used for this experiment are the SSSA robots developed as part of REPLICATOR projects. One of those robots is depicted on Figure 2.1. This robot has 8 wheels which can be used to move it on flat ground. When this robot is alone, the wheels are its only means of movement.

The whole robot, which resembles small cube with 125.4 mm long edge, consists of two main parts. The main body which in rendered blue is the larger part, the smaller arm is rendered grey. All wheels are connected to the main body. The whole robot weights 1.2 Kg (this value is used by the simulation, actual mass of real robots can be slightly different). From this mass 1 Kg is the mass of the body with wheels and the movable arm weights 200 g.

The main purpose of this robot is not to move by wheels, wheels are only used to move robot closer to other robots of same kind and to connect with them. For this purpose there are four slots on the robots body, they are on the main body of the robot (each on separate

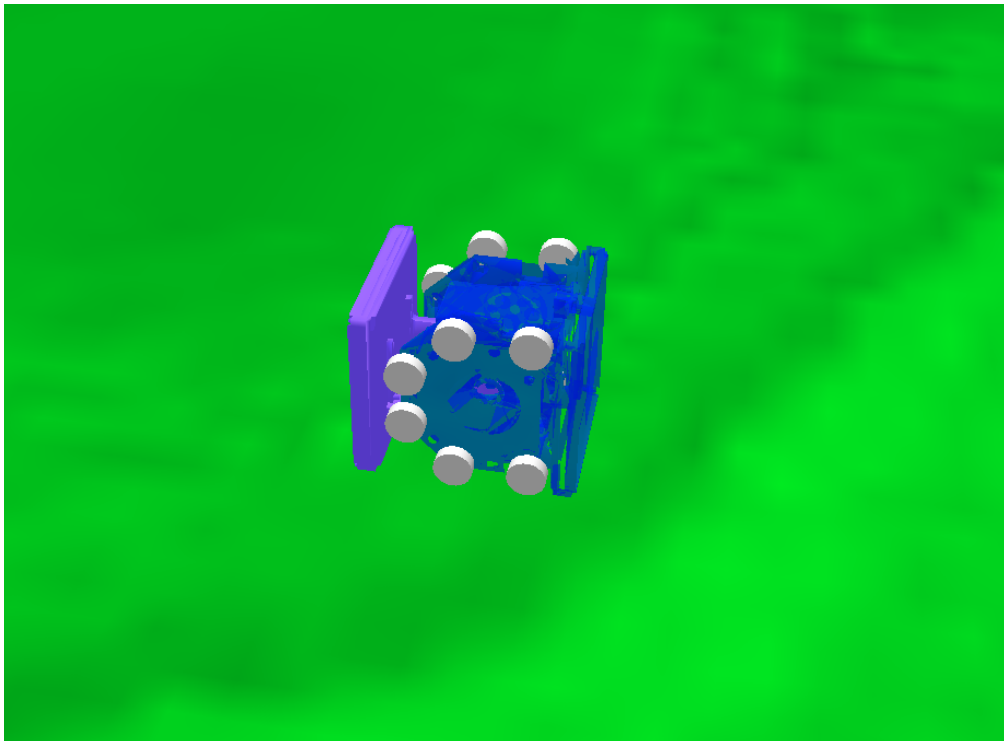

Figure 2.1: Detail view of a single robotic block used as a building block of the final robotic creature.

side) and one is on the movable arm. All of these slots can be connected to any slot on other robots. Once they are connected they stay fixed and can not be moved. All movements are managed by the robot's arm. This arm has its axle in the centre of the robot's body and can be moved between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. This allows moving the slot on the end of the arm between three sides of the robot, which means that all six sides of the cubic robot can be covered by slots. Additionally it is not necessary to connect to slots when they have the same orientation, it is possible to connect them when they are rotated by $\frac{\pi}{4}$, $\frac{\pi}{2}$ or $\frac{3\pi}{4}$ relative to each other. This allows for a wide variety of creatures to be created with these robots.

When the arm movement is not desired, it is possible to fix the arm to stay in one position, otherwise the arm is loose and can be easily moved. Since the fixation of the arm is done by the same electric motor that moves it, the fix has limited rigidity and it also drains the batteries of real robots. Limited power is not a problem in the simulation, but the limited strength of electric motors is. Electric motors of real robot should be able to produce force of 50 N. In simulation it is possible to use much higher numbers to allow for stronger arms.

In this work robots were always connected into larger organism and wheels are never used to move individual robots around. In fact most simulations were done with a simplified version of robots without wheels and incapable of individual movement. Movement and cooperation of individual detached robots is a non-trivial problem which is not part of this

work. Detailed robots with wheels also take much more processing time to simulate and later to render.

2.3 Description of Complete Robot

For all experiments a robotic creature assembled from robots described in Section 2.2 has been used. This compound robot can be seen in Figure 2.2, it consist of 17 small blocks connected to each other.

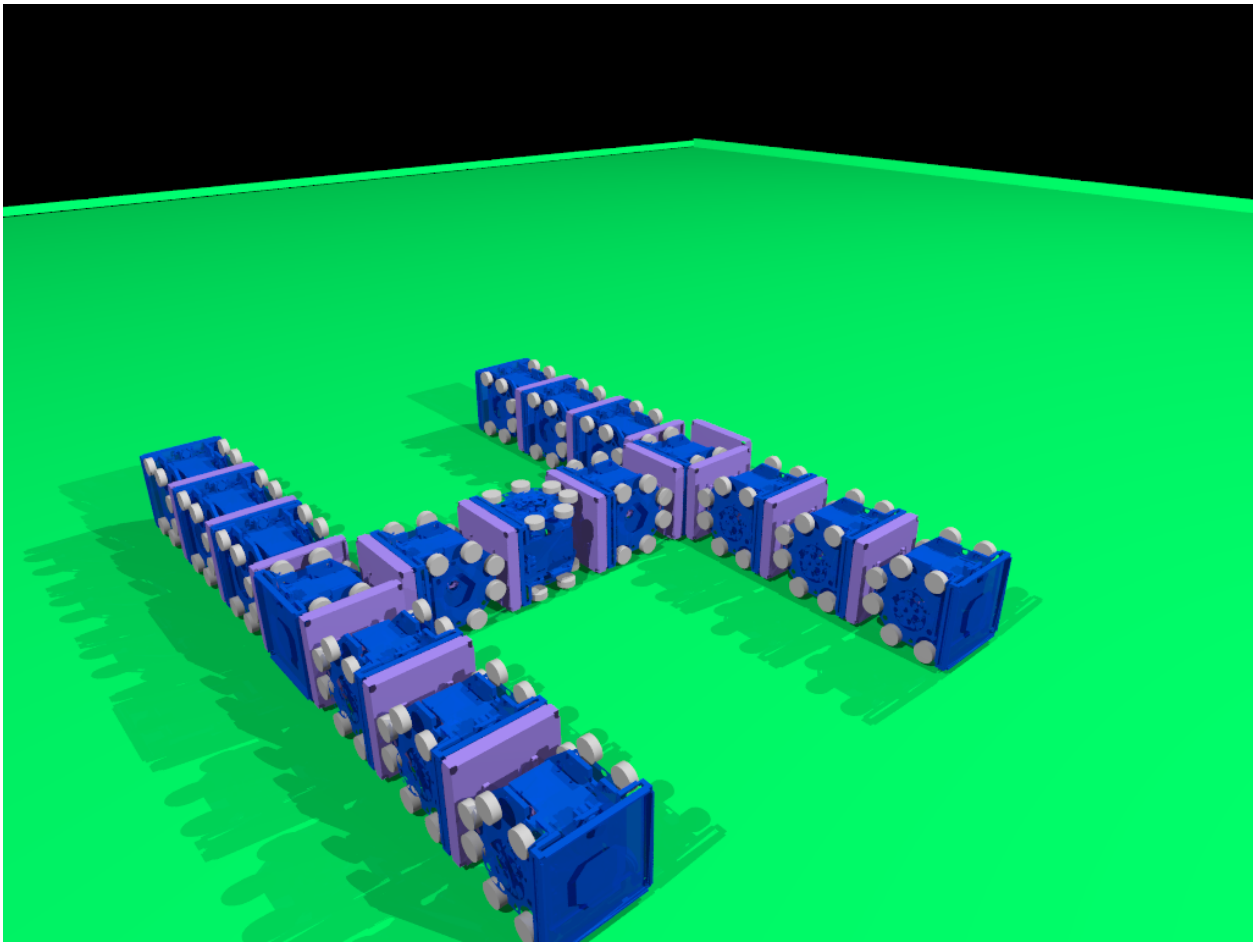

Figure 2.2: Complete robotic creature rendered with all details prior to any movement.

Those form a robotic creature with 5 block long torso remaining 12 blocks are used to form 4 legs, 3 blocks on each one. With one block 12.5 cm long this robotic organism is actually over 62 cm long with maximal leg-span of almost 88 cm. The whole robotic creature also weights 20.4 Kg. As it turned out the original electric motors with force of 50 N were unable to properly support an organism of this complexity in some less stable position. To increase the possibility of learning more cultivated motion patterns this number was increased to 200 N.

As can be seen in Figure 2.2 the central block of the torso is rotated by $\frac{\pi}{4}$ along the lengthwise axis. This configuration has three reasons. The first one is to allow for wider variety of movements of the torso. All SSSA robots can only move in one degree of freedom, should they all be connected with the same orientation an imagined spine of the robotic organism would only bend in one direction.

Rotation of one building block allows the creature to bend its spine both horizontally and vertically. Secondly the robot arm hold is always much stronger in the direction in which it can not move but it has a little allowance in the direction in which it can move. So the rotation of the central block strengthens the trunk in vertical direction at the cost of making it little more elastic in the horizontal direction.

The third reason for this configuration is the way how coupling sockets and arm fixation work. As has already been said the robot arm is fixed by the electric motor of the robot, this means that when large enough force is applied the motor will not hold and the arm can be moved. Unfortunately the coupling sockets have similar characteristic. When large enough force is applied to the coupling socket, it fails and the robot pulls apart. However the force needed to open the coupling socket is much greater than the force needed to move the robotic arm, even with the increased motor output. Therefore the rotated central block can act as a dumper and absorb some shocks passing through robot torso as it walks thereby preventing opening of coupling socket and breakdown of the robotic body.

The position of robot's arm can be easily brought back to desired position once the distribution of forces changes, the interlocking of robots is a much more complicate task. In fact interlocking of robots would not be even possible in some simulation related to this work. This is not just because robots are simulated only as simplified approximations without wheels, but because some robot configurations can not be achieved even with wheels. The central block of robot's torso is a fine example since it is laying on side and is clearly incapable of any independent movement. All robotic organisms in this work are assembled before the simulation starts.

There are in fact two slightly different robotic organism with the same shape. They can be easily distinguished in Figure 2.3. The left one is the same as the one on Figure 2.2, only standing and rendered with simple representation. The second one has the same general shape but there is a difference in the orientation of first segments of all legs (first segment being the one closes to the body). Those segments are rotated alongside the crosswise axis. This allows for limbs to move forwards and backwards. A short video of both robots can be found on attached CD-ROM in directory *Video* in files *Robot01.avi* and *Robot02.avi*.

The robots seen in Figure 2.3 represent those from Figure 2.1. This simplified version has almost identical behaviour in simulation but it can by calculated and rendered many times faster then the detailed one. In all following simulations, only simplified representation of robots is used.

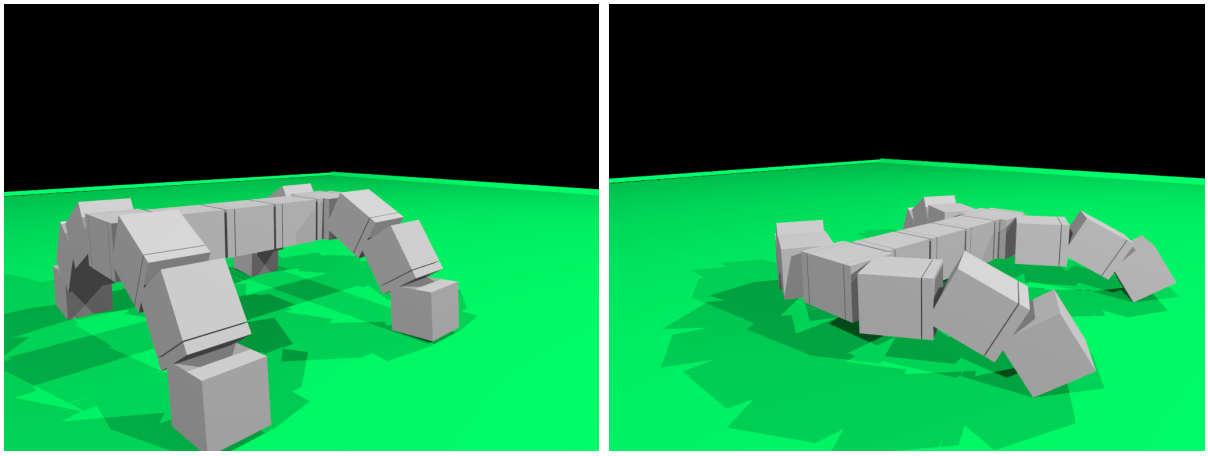

Figure 2.3: Two complete robotic creatures rendered in standing positions.

2.4 Project Goals

The main objective of this work is to design an algorithm that will be able to produce viable motion patterns for supplied compound robotic creature. There will be no attempt to automatically design topology or shape of those creatures and both creatures used in experiments are designed manually. The design of those robotic creatures is not random though. As the algorithm is expected to produce motion patterns similar to those seen in nature as those are believed to be very efficient, both robotic creatures were modelled in a way which allows animal-like motion. The creature depicted on the left side of Figure 2.3 is modelled to allow for motion close to the one used by crab. The creature itself does not resemble crab at all since it has only 4 legs, but it is designed to walk sideways just as crabs do. The second creature from Figure 2.3 is meant for lizard-like motion. Again there is no head or tail to resemble whole lizard but the configuration of limbs allows for forward walking creature with legs on side of the body.

This objective is to be pursued with the use of Evolution Algorithms and especially Genetic Programming.

Chapter 3

Computer Simulations of Robots

As SSSA robots described in Section 2.2 are still in stage of experimental prototype, it would not be advisable to use real robots for the evolution. For the start it is not an easy task to put together 17 of them and even if it would be, the use of physical robots has many disadvantages. For example they can break, they run on batteries which need to be manually replaced or charged and in general a lot of human supervision is needed. It is also slow as robots can only run at limited speed and while parallel execution of experiments is possible it requires more robots and lot of space. Last but not least, SSSA robots currently do not satisfy the requirements for this experiment as their motors have limited output.

For those reasons all experiments are conducted in a simulated environment. To simulate modular robotic organism its necessary to have a system capable of calculating physics of the simulated world.

The task of the physical simulation is to move the physical objects according to forces applied to them considering interactions between the objects and the environment. Two main approaches are used for physic-based simulation of rigid bodies: second Newton's law $F = m \cdot a$ or an impulse-based approach, where m is the mass and a is the acceleration of the objects. In this approach, first the forces are applied on the objects and acceleration of the object is computed. The movement of the object is then computed by integrating the acceleration. This approach is implemented in the widely used libraries ODE and Bullet.

3.1 Physical Simulation Algorithm

The pseudo-code of the physical simulation is listed in Algorithm 3.1. Before the main loop of the simulation can start it is necessary to initialize the world and all robots. The world is created in `init_world()` procedure and this includes also the initialization of the physical engine, variable definition (for example gravity and friction) and loading of the arena. The `init_robots()` creates the simulation models of all robots and places them to the specified positions in the prepared arena. The robots can consist of many movable parts whose orientation also needs to be set.

Algorithm 3.1 Simple Physical Simulation

```
1: init_world()
2: init_robots()
3: loop
4:     control_robots()
5:     collision_detection()
6:     physical_step()
7: end loop
```

After the initialization, the main loop can start and run as long as needed. The first procedure executed in every loop is `control_robots()`. It contains the code which applies forces to robot's joints and wheels in order to move them. Next the `collision_detection()` detects all interaction between the object in the arena and applies additional forces needed to stop them from merging (which would be impossible in the real world). Last procedure in the main loop is `physical_step()`. Here, all forces caused by the user control or interaction of bodies are applied to the objects of the arena and their acceleration is calculated. The acceleration is then integrated and new positions of objects are calculated. As the integration is only a numerical approximation it can have significant influence on simulation precision and stability. The integration time Δt is usually set to tens of milliseconds, shorter times causes simulation to be too slow, when it is too long, objects can obtain large velocities due to numerical errors.

3.2 Modelling a Robot and an Arena

In physical simulation, the simulated objects have several properties defined. For example a shape of the object, mass, position, rotation, translation and velocities. For the physical simulation itself, the shape of the objects is not important. However, it is used in the collision detection phase to determine how bodies interact during collisions.

The robot usually consists of several parts (main body, wheels, ...), these parts are usually connected by joints. Several types of joints can be defined, e.g. socket and ball, hinge or piston. The joints define the relative positions between the objects, which has to be preserved during the simulation. Joints also allow to define additional parameters like rotation velocity and moment of the joint. This can be used to simulate a motor using piston rotation.

The same primitives used to model the robots can be used to model the arena. All object can be divided into two categories. Dynamic, which are simulated in the same way the robots are but can not be controlled and static. Static object are movable and are not subject to gravity or other forces. This has the advantage that they do not need to be considered by the physical engine during the integration. This simplifies and speeds up the whole simulation.

3.3 Robot Simulator

The physics simulator while perfectly capable of simulating robots is a general purpose tool and does not include any functions specific to robot simulations. There are no predefined robots or arenas and robot control is rather difficult since it has to be done on lowest level. The visualisation is usually not a part of a physics simulator. To add the visualization and simplify the task of simulation initialization, a specialized robot simulator has to be used. It can also simplify the robot control by allowing the speed or angles to be set to the robot's wheels and legs. In this work *Sim* simulator has been used in all simulation [16].

Chapter 4

Genetic Algorithms and Genetic Programming

4.1 Genetic Algorithms

Genetic algorithms (a sub-discipline of evolutionary computing) are optimization heuristics inspired by Darwin's theory of evolution and natural selection. They are often described as global search methods that do not use gradient information. Thus, non-differentiable functions as well as functions with multiple local optima represent classes of problems to which genetic algorithms may be applied. Genetic algorithms, as a weak method, are robust but very general [19]. This makes them a perfect candidate for solving black-box types of problems.

Evolutionary strategies were invented by I. Rechenberg in the 1960s. Later, his ideas were developed into Genetic algorithms (GA) by John Holland. His book "Adaptation in Natural and Artificial Systems" [7] was published in 1975.

4.1.1 Canonical GA

The first procedure needed for execution of any genetic algorithm is an initialization of population. In this step, random individuals are generated until their count is the same as the desired size of the first generation. Those individuals represent solutions of the problem. The data representing the problem solution are often called genotype or chromosome. The next step is to use each individual as an input for a fitness function and assign a fitness value to it. Fitness can be generated directly by a fitness function or it can be based on the rank of an individual in the population [20].

4.1.1.1 Selection

When a fitness value is assigned to all individuals, parents for new generations can be selected based on this value. There are various methods that can be used for the parent selection, the most common are *Roulette Wheel Selection*, *Tournament Selection* or *Stochastic Universal Sampling* [5, 1] Their main purpose is to find the ideal candidates for mating.

4.1.1.2 Crossover

When parents are selected the algorithm progresses to the crossover. In this process parents are recombined into offspring and a new generation is created. The are many different methods of recombination for different representations of various problems. The most common crossover operators for problems represented as a binary string are the single-point crossover, the two-point crossover and the uniform crossover.

4.1.1.3 Mutation

After crossover a mutation operator is applied with a certain small probability. This operator also depends on the problem and the representation. It's main purpose it to introduce new or reintroduce lost genes into the population.

4.2 Genetic Programming

Genetic programming is a sub-discipline of genetic algorithms. It was introduced in 1990 by John Koza [11]. The goal of genetic programming (GP) is to automatically evolve computer programs the same way normal GA evolves solution to other problems. Therefore the basic structure and design of the algorithm stays the same as described in Section 4.1.1. The most important changes are in the representation, the crossover operator and the mutation operator.

4.2.1 Representation

Genetic programs have been traditionally represented as tree structures (although linear approach also exists [3]). This tree structure contains functions from function set F in internal nodes and terminals from set T in leaf nodes.
The functions in the function set may include:

- arithmetic operations (+, -, *, etc.),
- mathematical functions (such as sin, cos, exp, and log),
- Boolean operations (such as AND, OR, NOT),
- conditional operators (such as If-Then-Else),
- functions causing iteration (such as Do-Until),
- functions causing recursion, and
- any other domain-specific functions that may be defined.

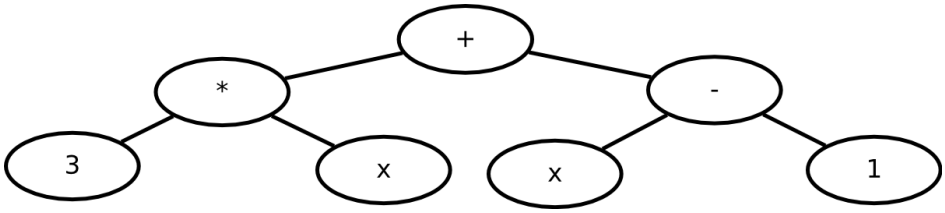

Figure 4.1: Typical individual in Genetic Programming.

Terminals are mostly just variables (may be input sensors and state variables) and constants [12].

An example GP individual is in Figure 4.1. This tree represents a simple arithmetical expression $3x + x - 1$ or `(+ (* 3 x) (- x 1))` in LISP syntax.

Virtually any programming language is capable of expressing and evaluating the compositions of functions described above (e.g. PASCAL, FORTRAN, C, FORTH, LISP, etc.) [11], but functional programming languages have the advantage of having a more convenient access to the program parse tree.

4.2.2 Initialization

As GP uses representation with arbitrary structure, some method of generating initial population is needed. Koza describes three methods of population initialization, namely *full*, *grow* and *ramped-half-and-half* [12]. Other methods for initializing population also exists as well but those don't improve fitness results compared to those three [13].

4.2.2.1 Full Method

The *full* algorithm generates trees with distance from root to every leaf equal precisely to the maximal allowed depth. This is accomplished by restricting the selection of the label for points at depths less than the maximum to the function set F, and then restricting the selection of the label for points at the maximum depth to the terminal set T [12].

4.2.2.2 Grow Method

The *grow* method generates trees with distance from root to any leaf node no greater than the maximal allowed depth. This is accomplished by making the random selection of the label for points at depths less than the maximum from the combined set $C = F \bigcup T$ consisting of the union of the function set F and the terminal set T, while restricting the random selection of the label for points at the maximum depth to the terminal set T. The relative number of functions in the function set F and the number of terminals in the terminal set T determine the expected length of paths between the root and the endpoints of the tree [12].

4.2.2.3 Ramped-Half-and-Half Method

The *ramped-half-and-half* is a mixed method which combines both *grow* and *full* method. With this method, half population is created with grow method. The other half is created by full method with tree depth set gradually from 2 to maximal allowed depth. The ramped half-and-half generative method produces a wide variety of trees of various sizes and shapes [12].

4.2.3 Selection

In genetic programming, same selection mechanism as ones used in conventional genetic algorithms can be used. There is no need to invent special selection operators just for GP, but one enhancement called Greedy Over-selection can lead to faster evolution and better results [12].

4.2.3.1 Greedy Over-selection

To reduce the number of generations required for GP to run, greedy over-selection can be used [18]. Over-selection is implemented by dividing population into two groups, a small group of the fittest individuals and a larger group of the less fit individuals. Then 80% of the time individuals are selected from the group of the better individuals and 20% of the time from the other group. Inside those groups, a normal selection process is used. The 80%-20% split has no particular justification; it merely provides a convenient way of causing the greedy over-selection of the fittest [12].

4.2.4 Crossover

Crossover (recombination) is a sexual way of producing new individuals. It combines two or more individuals and at least one offspring is created. The process of producing new individuals from the previously selected parents can be described as follows:

At first, one crossover point is selected in each parent as illustrated by Figure 4.2, this can be any leaf, an internal node or even the root node. The offspring is created by removing the sub-tree at the crossover point from one parent and replacing it by the sub-tree from the crossover point of the second parent. Another offspring can by produced by the same method from the second parent.

4.2.5 Mutation

Mutation is an asexual operation which produces one new individual from one parent. This is done by selecting a mutation point and replacing it by a new randomly generated tree. This is depicted on Figure 4.3.

In a conventional genetic algorithm, mutation has the function of reintroducing lost genes into the population and preventing premature convergence. In the genetic programming however there is a little chance of loosing one function since the function and terminal sets

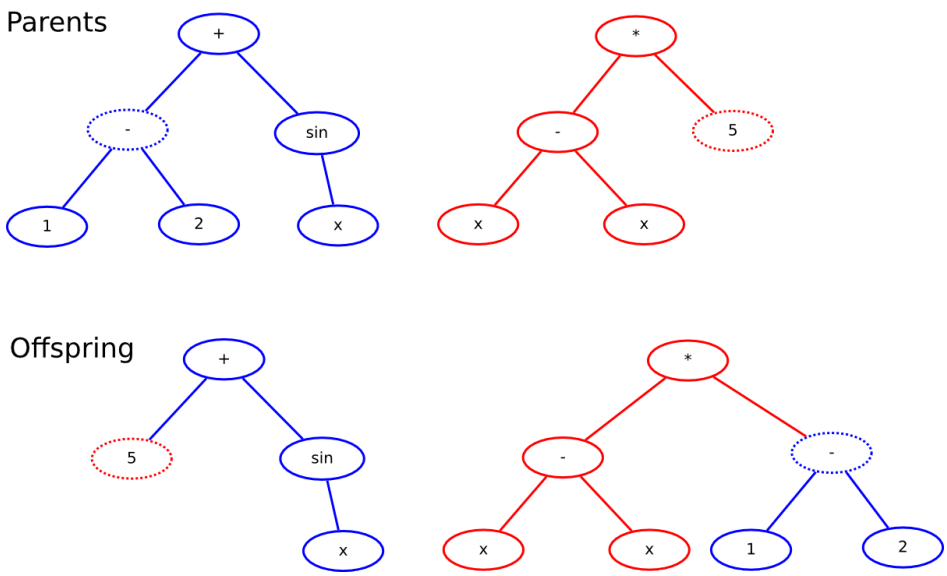

Figure 4.2: Crossover operation in Genetic Programming.

are much smaller then the number of positions where these functions and terminals can be used. Premature convergence problem is mostly solved by the crossover operator itself since it can produce new different individuals even when both parents are identical. For those reasons there is a little need for mutation in GP.

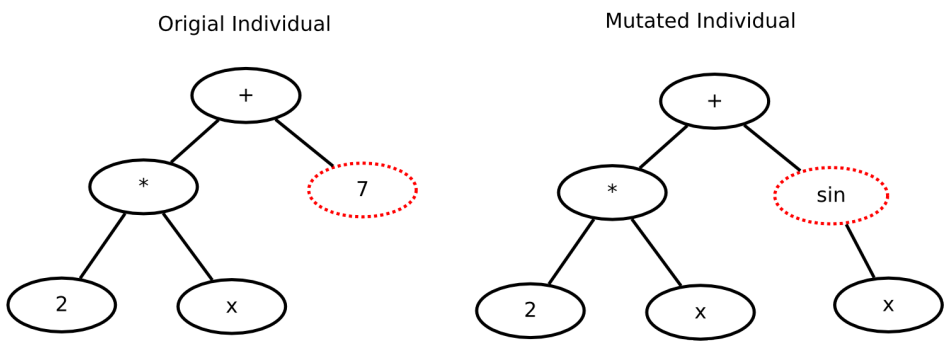

Figure 4.3: Genetic Programming mutation example, with marked mutation point.

Chapter 5

Analysis and Solution Proposals

The goal of this project is to create a system which can autonomously generate walking or other motion patterns for a given legged robotic creature. To achieve this the following system will be designed.
The core of the entire system consists of two evolutionary algorithms connected together to run in co-evolution as depicted on Figure 5.1. First EA implements genetic programming to evolve functions which will be used to control motors of the three segments of each leg. Let us call this algorithm EA-f (meaning EA-functions). The second algorithm is called EA-s (for EA-synchronisation) and it is a Genetic Algorithm which evolves a set of parameters used to modify the pattern evolved by the EA-f algorithm before they are copied to all four legs of the robotic creature. Final motion patterns are created by combining individuals from both algorithms. EA-s and EA-f needs to exist in cooperative co-evolution because only a combination of individuals from both populations can be evaluated by the *sim* simulator.

The reason behind this configuration is based on observation of how real animals walk and how their left legs do same motion but not in the same phase. Therefore instead of evolving complex system where there is a dedicated control function for every segment ot the robotic body, it should be possible to produce good-quality results by breaking this system into two smaller ones as described above.

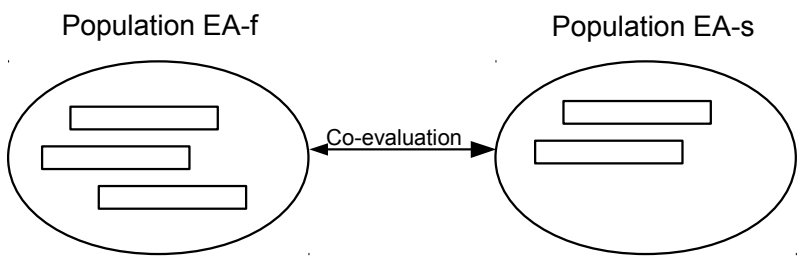

Figure 5.1: Schematic representation of two populations.

This chapter describes all operators and their settings used for both EA-s and EA-f algorithms.

5.1 Genetic Programming Algorithm

The first algorithm is a Genetic Programming set to evolve control functions for individual segments of a single leg.

5.1.1 Representation

Each leg in a robotic organism can be compound of multiple segments. Those segments serve as joints would in a real life organism and every one of them should have a control function. All those functions are evolved together in a single EA-f individual. In other words every individual in EA-f population contains multiple genetic programs, one program for every segment of the leg. Individual genetic programs use the same representation as the one described in Section 4.2.1.

5.1.2 Evolutionary Model

This EA uses generational evolutionary model. This is quite a common model with discrete generations of fixed size. The operation of this model is described by Pseudo-code 5.1.

In this model parents are selected from the old population with replacement and their offspring are placed into a new population until it has the same number of individuals as the old one. Once the new population is complete it replaces the old population and the whole process can begin anew. This means that children can never interbreed with their parents, it also means that all individuals are deleted after one generation of GA even if they are better than their offspring.

Algorithm 5.1 Generational Evolution

```
Create initial population.
loop
   repeat
      evaluate all individuals
      select parents from old population
      create offspring from selected parents
      insert offspring into new population
   until new generation is complete
   replace old population with a new one
end loop
```

5.1.3 Population Initialization

Before the genetic programming algorithm can start, it is necessary to initialize the first generation of individuals. This task would be simple in a normal genetic algorithm where the configuration of chromosomes is predefined and unchangeable, but individuals in GP are trees with an unequal number of nodes. To generate initial population in this algorithm, the "ramped half-and-half" method described in Section 4.2.2 has been used with maximum initial tree depth equal to 5.

5.1.4 Parent Selection

To select individuals for reproduction, the tournament selection method has been chosen. It is described by Algorithm 5.2. The idea is simple. Choose a number of individuals randomly from a population (with or without replacement), select the best individual from this group for further genetic processing, and repeat as often as desired [6]. The tournament size in this algorithm is set to 7.

Tournament selection can be easily combined with greedy over-selection (as described in Section 4.2.3.1), but greedy over-selection has not been used since it is only recommended for much larger populations than those used in this experiment [12].

Algorithm 5.2 Tournament Selection

1: **repeat**
2: select k (tournament size) individuals at random
3: select the best individual out of k individuals
4: **until** specified number of parents is selected

5.1.5 Crossover and Mutation

Both crossover and mutation operators for genetic programming were described in Sections 4.2.4 and 4.2.5. Crossover is applied with a probability of 0.9 and resulting individuals are mutated with a probability 0.005.

5.1.6 Bloat Control

When evolutionary computation uses arbitrary-sized representations, often the evolutionary process drives not only towards the fitter individuals, but often towards the dramatically larger individuals. This rapid increase in size, known as bloat can hinder the evolutionary mechanism itself and can slow successive generations to the point that further progress is not feasible [15].

To reduce the bloat and increase the computational efficiency of the algorithm, a bloat control method called Proportional Tournament is used.

The proportional tournament algorithm selects an individual using the tournament selection as usual, using some fixed tournament size S. However, the proportion of tournaments is selected based on parsimony (that means smaller trees) rather than on fitness. A fixed parameter R defines the proportion, where higher values of R imply more of an emphasis towards fitness: R = 1 implies that all tournaments will select the parents solely based on fitness, while R = 0.5 implies that tournaments will select on fitness or size with an equal probability [14].

5.2 Leg Synchronization

This is the second EA of the system and its purpose is to evolve synchronization of movements of robots legs. Individuals from this population can be used to generate valid motion patterns in combination with any individual from EA-f population described in Section 5.1. This EA is implemented in two alternative versions which differ in representation, crossover and fitness. Both implementations can be used alternatively as both implement the same interface, but the rational coded one is used in most experiments.

5.2.1 Representation

5.2.1.1 Rational Coded Representation

The chromosome of each individual in this implementation consists of two arrays with the same number of fields as is the number of robots legs, in this case 4. First array contains boolean variables that indicate whether motion patters should be used in forward or reversed direction in the corresponding leg.

The second array contains values of the motion phase shift for each leg. Those values are represented as fractions with lowest possible value of 0 and largest possible value 1. To further reduce the size of the search space, those fractions are rounded so that their denominator is always smaller than 12.

5.2.1.2 Binary Coded Representation

In this implementation of EA-s all information is encoded in a single array of binary values. For each leg, there is one bit representing direction of motion and ten bits for phase. Phase values can range from 0 to 1023, but the number is divided by 1024 to produce values between 0 and 1. With total of four legs and eleven bits for each one, the whole array is 44 bits long. Values for single leg are grouped together, first is the direction bit and the ten bits for phase follow. The order of legs is forward-left, forward-right, back-left and back-right.

5.2.2 Population Model

This entire population is evolved with the steady state evolution model. This model is described by Algorithm 5.3. There is always only one population and offspring are placed

directly between their parents and can immediately start breading with them. Constant population size is maintained by removing the worst individual from the population every time a new one is added. It would be possible to devise another replacement scheme but replacing the worst is the most widely used and almost always a sufficient policy [17]. It also leads to fast convergence, that is beneficial for out purposes.

Algorithm 5.3 Steady State Evolution

```
1: Create the initial population.
2: loop
3:     select parents
4:     create offspring from selected parents
5:     add offspring to the population
6:     remove worst individual from the population
7: end loop
```

The reason for this choice is the way both populations are linked. Should this population use a more common generational evolution model all connections from GP individuals in EA-f would be lost after every generation of EA-s and it would be necessary to search for new links for every individual, which is a computationally intensive task with questionable justification. However when the steady state model is used, most links are preserved and broken links can be replaced more efficiently.

5.2.3 Selection

To select individuals for crossover tournament selection is used. The Pseudo-code 5.2 for this selection operator has been already described in Section 5.1.4, the only difference is that this time tournament size is set to 3.

5.2.4 Crossover

5.2.4.1 Crossover for Rational Representation

As chromosomes of individuals from this population consist of two arrays one with binary values and one with real numbers (as described in Section 5.2.1) it would be quite inefficient to perform crossover operation with a single crossover operator. Therefore crossover is divided into two separate parts.

Firstly uniform crossover is used on the first array. As it contains only binary values not necessarily sorted in any purposeful order this crossover operator should be a good choice.

A modified linear crossover [22] is used for the second array. With this operator an offspring is created from two parents. Let p_1 and p_2 be genes from parents from the same position in parent chromosomes, then parent gene on the same position will be a random choice from a set of values computed as $\{1.5 \cdot p_1 - 0.5 \cdot p_2, p_1, 0.5 \cdot (p_1 + p_2), p_2, 1.5 \cdot p_2 - 0.5 \cdot p1\}$ rounded to admissible numbers.

5.2.4.2 Crossover for Binary Representation

A simple single-point crossover has been used with this representation. This crossover takes two individuals, and cuts their chromosome strings at some randomly chosen position, to produce two "head" segments and two "tail" segments. The tail segments are then swapped over to produce two new full length chromosomes (see Figure 5.2). The two offspring each inherit some genes from each parent [2].

Figure 5.2: A Single-point Crossover

5.2.5 Mutation

5.2.5.1 Mutation for Rational Representation

Mutation operator for rational representation is selected according to the parameter to be mutated. First array is mutated the same way ordinary binary-represented chromosomes are, that is by switching value of a single bit. When a number from the second array is selected for mutation, it is replaced by a new randomly generated value.

5.2.5.2 Mutation for Binary Representation

Mutation is applied to offspring chromosomes after crossover. In this case mutation changes value of one gene from True to False or vice versa with the probability of 0.005.

5.2.6 Replacement

As this population is evolved using steady state population model, it is necessary to choose method of inserting new individuals into the population and removing the old ones. Replacing the worst individual is the most widely used and almost always sufficient policy [17].

5.3 Population Connection and Co-evolution

A co-evolution part of this experiment is the most important one since individuals from neither population can control the robot and be evaluated only by themselves. For co-evolution to work properly, some important parameters have to be determined and set.

5.3.1 Link between GP Trees and Parameters

To control the whole robot, two individuals, one from each population, are necessary. It's not possible to pair every one individual from EA-s population with only one individual from EA-f population, since the EA-f population is larger. EA-f population is also almost entirely replaced in every generation and later in the run of the algorithm most individuals from EA-f achieve the best results with the same one or just a few individuals from EA-s. Therefore the best way to connect populations is to assign one individual from EA-s population to each individual from EA-f population. During initialization this assignment is done randomly, later local search for better combinations of individuals is performed every 20 generations. The best possible link between EA-f and EA-s is always selected as this optimistic approach is generally the best method for collaborative fitness assignment [21].

5.3.2 Evolution Synchronization

Since both populations are of different size and one uses discrete generations while the other is steady state, it is necessary to set some relative speed at which both population will be evolved. While it would be possible to devise some sophisticated dynamic system to calculate this parameter, a simple manual setting can produce good enough results and allows for settings which will allocate computational resources more efficiently.

As it turns out there is a little or no improvement in EA-s algorithm during the first 10-20 generations of EA-f algorithm. To save some computational time with little or no effect on performance, EA-s population is just filled with random values and its evolution starts after 20 generations of EA-f algorithm. Then, every two generations, 30 new parameter individuals are created. This number is a trade-off between performance and speed of the algorithm. Larger number of new individuals can produce slightly better results, but the necessary resources are quite inadequate.

5.3.3 Fitness

No individual from EA-f nor one from EA-s can control the whole robotic creature without the cooperation of one individual from the other EA. Therefore fitness is always calculated for a pair of individuals, one from each EA. The *sim* simulator is used to calculate this value. The objective of the robot is always to reach some point in space by one segment of its body. The exact location of this point in space vary between different experiments, but fitness is always calculated as the shortest distance from that point to selected body segment.

When the fitness is calculated it must be assigned to both participating individuals. This is simple in case of EA-f individual because every EA-f individual always has only one EA-s individual assigned. Therefore fitness of the whole pair can be easily assigned to participating EA-f individual.

This is not so simple in case of EA-s individuals since those can be connected with multiple individual from EA-f or with no individual at all.

But some meaningful value or at least order of individuals is necessary for correct operation of Genetic algorithm. To provide some fitness to all individuals in EA-s the following scheme has been devised:
Each individual just after it is created is evaluated in combination with best, worst and several random individuals from EA-f. In this process the individual from EA-s population may or may not be linked with one or more individuals from EA-f population, either way the best fitness value is saved and whenever this individual is evaluated again, the new value is compared to the saved one and is replaced if it is better. Thanks to this each individual from EA-s population remembers the best fitness it has ever achieved, however this value may never be used as actual fitness.

The final fitness used by GA is calculated as follows:

- If the parameter individual is linked with single individual from EA-f population, then the fitness of this linked pair is used.
- If it is linked with more than one individual from EA-f population, then the best fitness from all linked individuals is used.
- Last possible scenario is that there is no link to any individual from EA-f population. In this case, the saved fitness value is used but first some penalty is added. This way Individuals with no link to EA-f can still compete between each other but will always loose against those that are actually useful. This is especially important when replacement occurs, because useless individuals are replaced first and there is less need to search for parameters for orphaned individuals from EA-f.

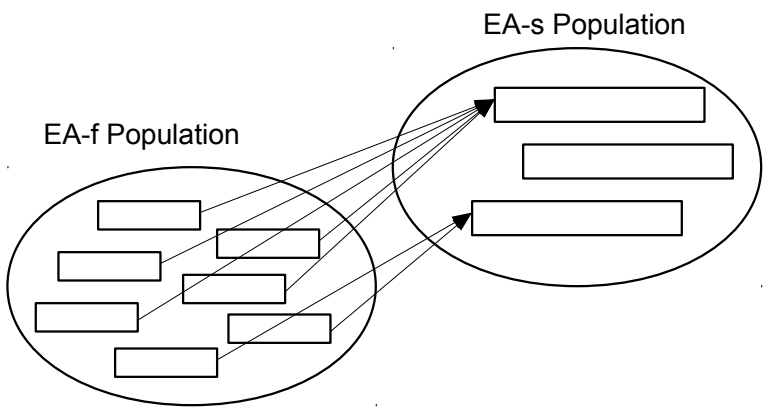

Figure 5.3: Detailed schematic representation of two populations.

Chapter 6

Implementation

6.1 Language and libraries

This section discuses programming languages libraries and other tools used in the implementation of the experiment.

6.1.1 Programming Languages

The first objective that needs to be finished before the implementation of the proposed algorithm can begin is the selection of programming languages and other tools and libraries suitable for the task. To successfully implement Genetic Programming algorithm a programming language needs to be able to manipulate computer programs as data. This programming language also needs to be able to compile and link those programs or support an interpreter to execute them.
For many task in evolutionary computing, it is also advantageous when a programming language has some high-level list manipulation facilities.

Virtually any programming language satisfies those requirements but there is another element specific to this problem and that is the need to use C++ library for fitness evaluation. This proves to be very important factor since not all languages can easily call C++ functions. The obvious choice of C++ language has been ruled out because it lacks some modern features like a garbage collector which is very useful for a GP implementation. It would also make the whole implementation large and rather difficult without any relevant advantage.

From other languages, Python has been chosen primarily because its extendibility by C and C++ code is one of its core features. It also features optional garbage collector and has many features suitable for EAs.

6.1.2 Python

Python is a high-level, object oriented, general-purpose, interpreted programming language. It has a feature set very similar to that of LISP with the main difference being the absence

of macros. Its fast implementation of lists and availability of lambda calculus makes it good choice for genetic algorithms and genetic programming. The only disadvantage may be the speed. As Python is an interpreted language which can be only compiled into a byte code and not into a native machine code, its execution can be in extreme cases several times slower than that of some natively compiled languages. Fortunately this is not a serious problem since the most computationally intensive parts of the experiment are written in C++. Python has been used to write the core classes and functions for the entire experiment.

6.1.3 C++

C++ is a free-form, statically typed, compiled, general-purpose programming language. It contains both high-level and low-level language features and there is no garbage collector in the core implementation. Its main advantage is the speed in certain tasks and a great number of available third party libraries. On the other hand it lacks some important or useful features, for example it has no native threading facilities in the standard library. The *sim* simulator is written entirely in C++ language and it also uses several C++ libraries for graphics and physics calculations. To initialize and run the robotic simulation some C++ classes had to be implemented as it would be unnecessarily large amount of work to write Python interface for the entire *sim* library.

6.1.4 Cython

Cython is a programming language designed to simplify writing of C modules for CPython Python interpreter. Cython is based on Pyrex but has additional functionality and uses some better optimization techniques. The Cython language syntax is a superset of Python syntax with additional support of direct calling of C and C++ functions and methods. It also supports optional strong typing of variables to allow for more aggressive code optimisations in frequently called functions. Cython source code compiles into C or C++ language which can be later compiled into a machine code, linked with additional libraries and loaded into a CPython interpreter.

Cython language has been selected to provide interface between C++ language (which was used for physical simulation) and Python language (which was used for evolutionary computations).

6.1.5 Libraries and Other Tools

6.1.5.1 Genetic Programming Tools

There are several Python libraries which implement genetic algorithms and genetic programming classes and functions. The most promising candidates for inclusion in this experiment were packages DEAP and Pyevolve.

6.1.5.2 Pyevolve

Pyevolve is a genetic framework written in pure Python. It is still under development but stable and usable. It can be easily extended to support new representations and operators but it lacks any initial support for genetic programming and co-evolution would have to be implemented from scratch. While it is certainly an interesting project its contribution to this experiment would be minimal.

6.1.5.3 DEAP

DEAP is an easy to use distributed evolutionary algorithm library written in the Python language. It consist of two main components, namely *Distributed Task Manager* (DTM) which distributes workload over a computer cluster and *Evolutionary Algorithms in Python* (EAP) which provides methods, data structures and tools for evolutionary programming. While this library has some important features like support for distributed computing, it is currently under intensive development and still lacks some features important for this experiment. The worst problem is that the distributed environment does not work with the *sim* simulator.

6.1.5.4 Conclusion

None of the tested libraries were suitable for the purposes of this work. In the end a custom solution had to be implemented to fulfil all requirements.

6.2 The Evolution Package

The EA part of the whole experiment, that is both EA-f and EA-s algorithms and all methods needed for their interaction are implemented in pure Python. Everything needed for proper evolution is part of the *evol* package and it is completely separated from fitness calculation code.

This package can be described as compound of three main parts

- EA-f experiment
- EA-s experiment
- Co-evolution code

Those three parts are connected to the *sim* simulator by the co-evolution code.

6.2.1 EA-f Algorithm

All the parts of the EA-f algorithm are in files *gp_population.py* and *functions.py*.

`GpRoot` class is the most important class of this algorithm. It is a derived class of the `GenericIndividual` class and instances of this class are the individuals of the EA-f algorithm. Pivotal attributes of this class are the list of control programs for individual leg segments and reference to one EA-s individual.

Methods implemented by `GpRoot` are amongst others mutation and crossover. Those methods call mutation and crossover methods in all control programs and construct new individuals from the returned results. Then there are methods for handling link to the EA-s algorithm and the fitness value. This link to EA-s algorithm and fitness value are very closely bonded and in most cases one can not be changed without the other.

At last there are methods for printing symbolic representation of the whole individual as LISP, Python or C function.

`GpNode` is the next most important class and it is used for all internal nodes, leaf nodes and the root node of all control programs. The main purpose of instances of this class is to act as a container for a single function or terminal, but it also manages all the functions needed for the evolutionary processes.

Crossover and mutation can be initiated from any node in genetic program and even though this may seem needles and redundant, there is a reason for this behaviour. In fact the crossover and mutation are only initiated from the root node of any program tree but crossover operation can result in any node of the tree becoming the new root of an offspring tree. For this reason it is much easier when every node contains code necessary to perform the crossover and mutation operations.

Crossover operation starts by selecting crossover point and asking the other parent for its crossover point, then the recursive copying of trees is initiated but with sub-trees with roots at cx-point being swapped. This operation preserves both parents unchanged so that they can be used again in another crossover.

Offspring produced by crossover can be mutated with some probability. If this operation should occur, it is again initiated from the root node of the selected tree. The operation starts by selection of a mutation point which is chosen randomly from all nodes of the tree with the same probability. The selected node is then replaced with a new randomly generated tree of the same maximal height as was the maximal height of the original sub-tree.

Aside from other methods necessary for proper working of crossover and mutation, there are also methods which provide printing and numerical evaluation facilities. Those methods work recursively and return a string with code or a numerical value of function for the whole sub-tree from the node at which they were called.

`GpParams` is a support class which is used by `GpNodes`. Every node in a single tree has reference to the same `GpParams` object. This object has access to some variables which are

used when a new tree is generated, like the list of function and terminal symbols available. This information is not only needed for initialization but also for mutation where new trees are also generated.

More importantly a complete set of references to all nodes in a tree is saved in `GpParam` objects, this is very important for both crossover and mutation operations since the whole tree does not need to be traversed every time one of these operations is performed.

The last part of the EA-f algorithm is in the file *functions.py*. This file contains definition of all functions and terminals used by the `GpNode` object. Those functions are in fact callable objects which is very convenient because an additional method can be implemented to ensure proper printing of the code.

6.2.2 EA-s Algorithm

The whole implementation of EA-s algorithm is in the file *param_population.py*. There is only one class called `ParamIndividual` whose instances are individuals of EA-s algorithm. This class is also derived from class `GenericIndividual`, same as `GpRoot` class which is described in Section 6.2.1.

The crossover and mutation methods for this algorithm are much simpler than those used in EA-f algorithm. There are two alternative implementations of them for two alternative chromosome representation. Both crossover methods are described in Section 5.2.4 and both mutations are in Section 5.2.5.

More interesting are the methods which handle links with individuals from the EA-f algorithm and fitness. Fitness is one reason why the EA-s individuals need to know all their links with the EA-f individuals. The reason for this is properly explained in Section 5.3. Each EA-s individual holds a set of references to the EA-f individuals, this set is continuously updated with new EA-f individuals while non-existent ones are deleted. This way any EA-s individual can easily extract its fitness whenever needed.

Another important function for which this set of EA-f individuals is very important is the deletion of EA-s individuals. That is because individuals in EA-f algorithm are by definition always connected to exactly one individual from EA-s algorithm. Therefore whenever an individual from the EA-s algorithm is deleted it is important to find a new EA-s individuals for all EA-f individuals which were connected to it.

6.2.3 Co-evolution Algorithm

The last but not the least important part of the evolution package is contained in the file *population.py*. This part is essential for proper evolution and cooperation of the EA-f and the EA-s algorithms. The central class of this part of `evol` package is named `Populations` and its main purpose is to hold both EA-s algorithm and EA-f algorithm populations. It

also has methods which can move both populations one generation forward and a method `write_stats()` which prints statistics of current generations into a file or standard output.

Aside from evolution code, there is a small function called `fork_sim()`. This function actually calls the *sim* simulator and manages fitness evaluation of all individuals. The interesting fact about this function is that it is never executed in the main process but is always sent to prepared process pool. This behaviour has two reasons.

First one is that the simulation run is the most computationally demanding part of the whole experiment and its execution takes significantly more time than the rest of the algorithm. Therefore it is absolutely necessary that this function can run in parallel whenever there is a hardware that allows it. In fact the whole experiment with all the simulation runs necessary would take so much time on single processor there is little reason to even attempt it.

The second reason is in the simulator itself. Due to some implementation details the simulation can not be simply reset to the initial state but its necessary to start it anew every time. Due to the way the CPython Python interpreter is implemented, it is not currently possible to properly reload modules which have already been loaded. This is actually the reason why the otherwise perfect DEAP library, described in Section 6.1.5.3, could not have been used. However the deployment of the process-pool solves the whole problem. The simulator module is only loaded to processes in the pool and when the simulation is done, the process is ended and replaced by a new one. This way every simulation can be run with a freshly loaded *sim* module.

6.3 Fitness Calculation

One of the most important parts of any evolutionary algorithm is the fitness evaluation of individuals. There are various ways how to calculate this value. It can be the result of a simple mathematical function or it can even depend on some real world process, this is actually not that uncommon method when robots are involved [4, 10]. But the use of physical robots to determine fitness has many disadvantages, especially the fact that it has to be done in real time which is often too slow.

Fitness of all individuals in both EA-f and EA-s populations is calculated with use of the *sim* robot simulator. This library is written in C++ language with the Open Dynamics Engine or the Bullet engine for physical simulations and Open Scene Graph for an optional visualization.

The *sim* simulator contains all functions necessary for physics computations and graphical representation of predefined robots. But no actual scenarios are prepared and must be created from scratch.

6.3.1 Simulation

To create a new simulation the `sim::Sim` class from the *Sim* library has to be inherited into a new class. This is illustrated by following code sample:

```
class S : public sim::Sim {
  public S() {
    initODE();
    setTimeStep(Time::fromMS(10));
    setTimeSubSteps(5);
    createArena();
    createRobots();
  }
}

int main(void){
  S sim;
  sim.run();
  return 0;
}
```

As can be seen, this simulation uses ODE engine for physical calculations. The method `initODE()` is in fact used to set parameters of the simulated world. Methods `createArena()` and `createRobots()` are used to add components of the robotic arena and individual robots into the world.

The code which generates the arena and places all the robots can be found in file *robotSimulation.cpp*. In this experiment arena is a flat surface with four low walls. Inside this arena there are 17 robots connected into a four legged robotic organism.

6.3.2 Arm Controller

All robots and parts of the arena are modelled using `components`. The components are objects inherited from `sim::Component` class and they are periodically called in each step of the simulation. An example of a simple component is in the following code:

```
class ExampleComp : public sim::Component{
  public
  ExampleComp();
  ~ExampleComp();
  void init(sim::Sim *sim);
  void cbPostStep();
  void cbPreStep();
}
```

Functions `cbPostStep()` and `cbPreStep()` are called after and before each step of the simulation. They need to be registered. This may be done in `init()` function as illustrated by following code:

```
void ExampleComp::init(sim::Sim *sim){
  sim->regPostStep(this);
  sim->regPreStep(this);
}
```

Those two functions are necessary to control the robot. All manipulations with the robot are done by calling the `reachArmAngle()` method of the robot every time it should move to another position. Second important task that has to be done from within those function is reading of the robots position, this is essential for fitness calculation.

6.3.3 Fitness Wrapper

Now when all parts of the experiment are described, the only thing that remains is to connect them together. Since the simulator and all related code are written in C++ while all evolutionary programming code was written in Python, a C++ wrapper for Python was needed. For this purpose a wrapper class has been implemented in Cython. It can be found in file *fitness.pyx*.

It is written in Cython programming language and it makes methods of class S described in Section 6.3.1 callable from Python. This file is compiled into C++ by Cython compiler. This C++ file is later compiled by gcc and linked with *Sim* simulator producing `.so` library which can be loaded into Python interpreter.

Chapter 7

Testing

To evaluate a functionality and efficiency of the implemented genetic algorithm several experiments were performed prior to the main experiment. All experiments were performed on the same simulated robot with some minor adjustments and their purpose was to evaluate functionality of different functions needed for a correct run of the final algorithm.

Most of the following experiments were executed with 500 individuals in EA-f algorithm population and 40 individuals in EA-s algorithm population. The duration of experiments was 200 generations of EA-f algorithm. Only exceptions are the first two experiments (7.1.1 and 7.1.2) which used only 160 individuals in EA-f algorithm and were terminated after 140 generations.

7.1 Experiment Set-up

There were two experiments performed before the first attempt to make the robot walk, then there were two experiments with differently configured walking robots and one final experiment focused on testing the EA-s algorithm.

7.1.1 One Leg Experiment

This experiment has been set-up with four legged robot constructed from 17 block. The same one as used in final experiment but with all but one leg disabled. Fitness function was also different as it was not to move the robot but only to reach a specified point with this one functional limb.

The main purpose of this experiment was to evaluate the functionality of the implemented genetic programming algorithm (EA-f) and successfulness of genetic programs controlling a single leg. The EA-s algorithm has been disabled for this experiment as its functionality was undesired.

7.1.2 Two Legs Experiment

This experiment is quite similar to the experiment 7.1.1, but this time two adjacent legs were enabled. The goal was for those two legs to touch above the main body of th robot. The purpose of this experiment was to determine if both EA-f algorithm and EA-s algorithm can co-evolve to accomplish a simple task that requires cooperation of more than one leg.

7.1.3 Four Legged Walking Robot

After the confirmation that all important parts of experiment work correctly, the next step is to connect all limbs and start the main experiment.

7.1.3.1 Sideways-walking Robot

This is the same robot as the one that has been used in all previous experiments, only this time all four legs are functional.

The target point is set beyond the left border of the desk on which the robot is placed so that it can not be reached without the robot moving its entire body. Fitness function is calculated as a distance of the centre block of the robot to the target point. To intensify some desired behaviour and discourage undesired one, this number is further modified.

Body Posture Because robots showed tendency to drag its body on the ground some penalty to the fitness is added whenever centre of body is below a specified height. This encourages the robot to hold it is body up and walk correctly instead of crawling on the ground.

Walking Direction Many robots showed tendency to drift to the left side or ti the right side or even work in circles. To reduce this effect distance in sideways direction is multiplied by a small constant.

Jump Prevention In first few generations of GA run, some robots gained advantage by making first long step or even jump but this often caused robot to fall over or went to another position which made another movement difficult or impossible.
To prevent such individuals from dominating the entire population and lead it to undesired local optimum, simple modification to fitness function was necessary. When the simulation is started, the robot is given small amount of time to make an initial step. After that time coordinates of the robot are read and added to target point. That way, all distance gained in the first step and possible fall is lost and robots which are capable of continual movement have advantage even if that movement is slow.

7.1.3.2 Forward-walking Robot

This experiment is very similar to 7.1.3.1, the main difference is in the configuration of the robot. The shape of the robot is preserved, but the first block (the one closest to the body) on each leg is rotated so that it no longer moves legs up and down but forth and back. Fitness is changed so that target point is no longer on side but in front of the robot.

7.1.4 Synchronization Experiment

In this experiment, the entire EA-f algorithm is replaced with a single individual from the Experiment 7.1.3.1 and only the EA-s population is evolved.

With fitness function set same as in Experiment 7.1.3.1 the goal is to confirm that evolution in EA-s algorithm works correctly and can produce usable parameters for given genetic program.

7.2 Experiment Results

In following parts ot the text, results of all experiments described in Section 7.1 are described and evaluated.

7.2.1 One Leg Experiment

This is the experiment described in Section 7.1.1. Its main purpose was to evaluate functionality of EA-f algorithms and suitability of GP for the task of controlling one leg. The results of this simple experiment are quite satisfactory as can be seen in file *Robot03.avi* on attached CD-ROM and important frames are also captured in Figure 7.1.

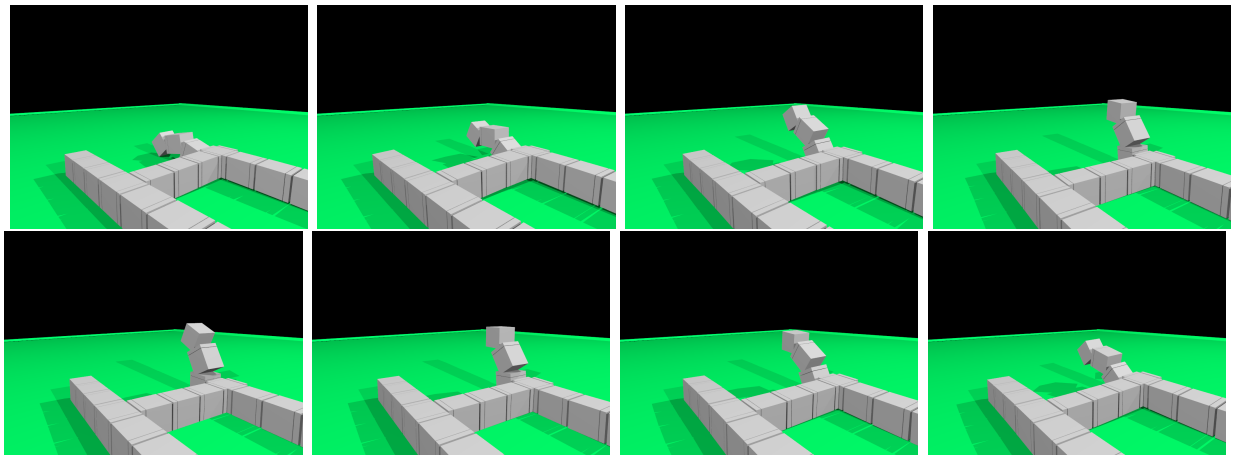

Figure 7.1: Movement of a robotic creature with one functional leg.

One interesting fact that can be observed from the video but is not clear from Figure 7.1 is the effect this simple motion has on the entire robot. It was expected that the stationary

robot body would provide immoveable base for the one moving leg, but as can be seen from the video, the whole robot shifts its position a little when the leg is moving. The significance of this effect can be clearly seen in Figure 7.2 where the trajectory of the last segment of the leg is recorded.

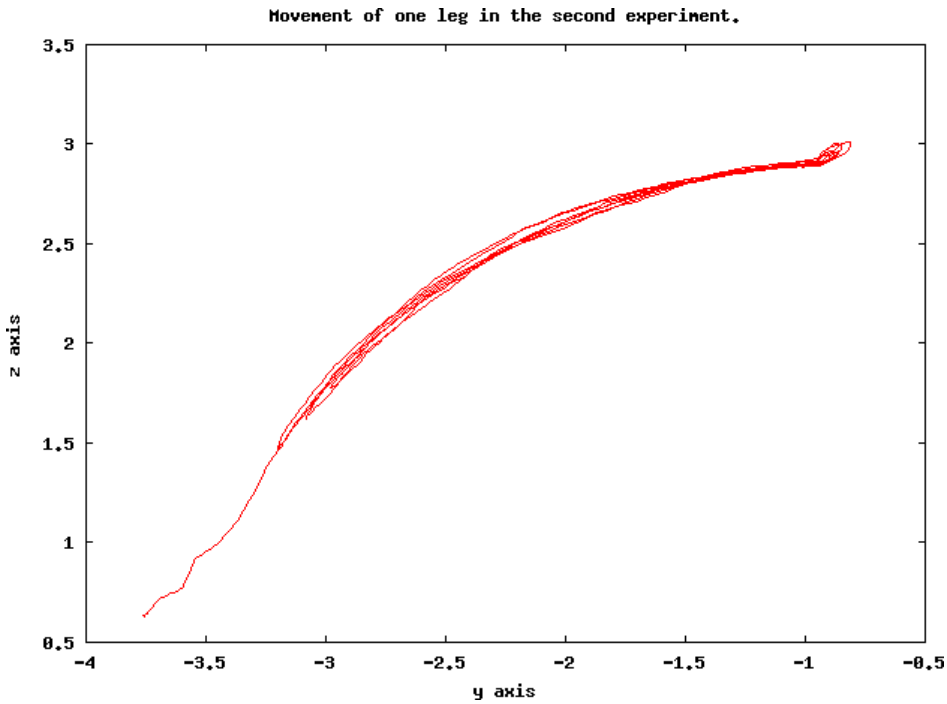

Figure 7.2: Trajectory of a single leg in the first experiment.

The target point which had to be reached is on coordinates <-1,3> and the robot managed to get closer than 0.001 units from this point, but only during one move. As the robot body shifts the trajectory of leg is slightly different.

However in spite of this unexpected effect this experiment has clearly shown that GP can be successfully used for control of this robotic organism.

7.2.2 Two Legs Experiment

The settings for this experiment is described in Section 7.1.2. This time two legs were moving with rest of the robotic body again serving as a base for those legs. The result of the evolution is recorded on video *Robot03.avi* in the same directory as the previous video files and part of the movement is depicted in Figure 7.3. It can be clearly seen from the video that the robot body again shifts its position, but this time it is not a problem since the target point is not absolute in space but it is relative to the robot body. The trajectory of the last segment of the left leg is recorded in Figure 7.4. This figure only shows movement of one leg, the second leg is controlled by the same program and its movements are identical, only mirrored.

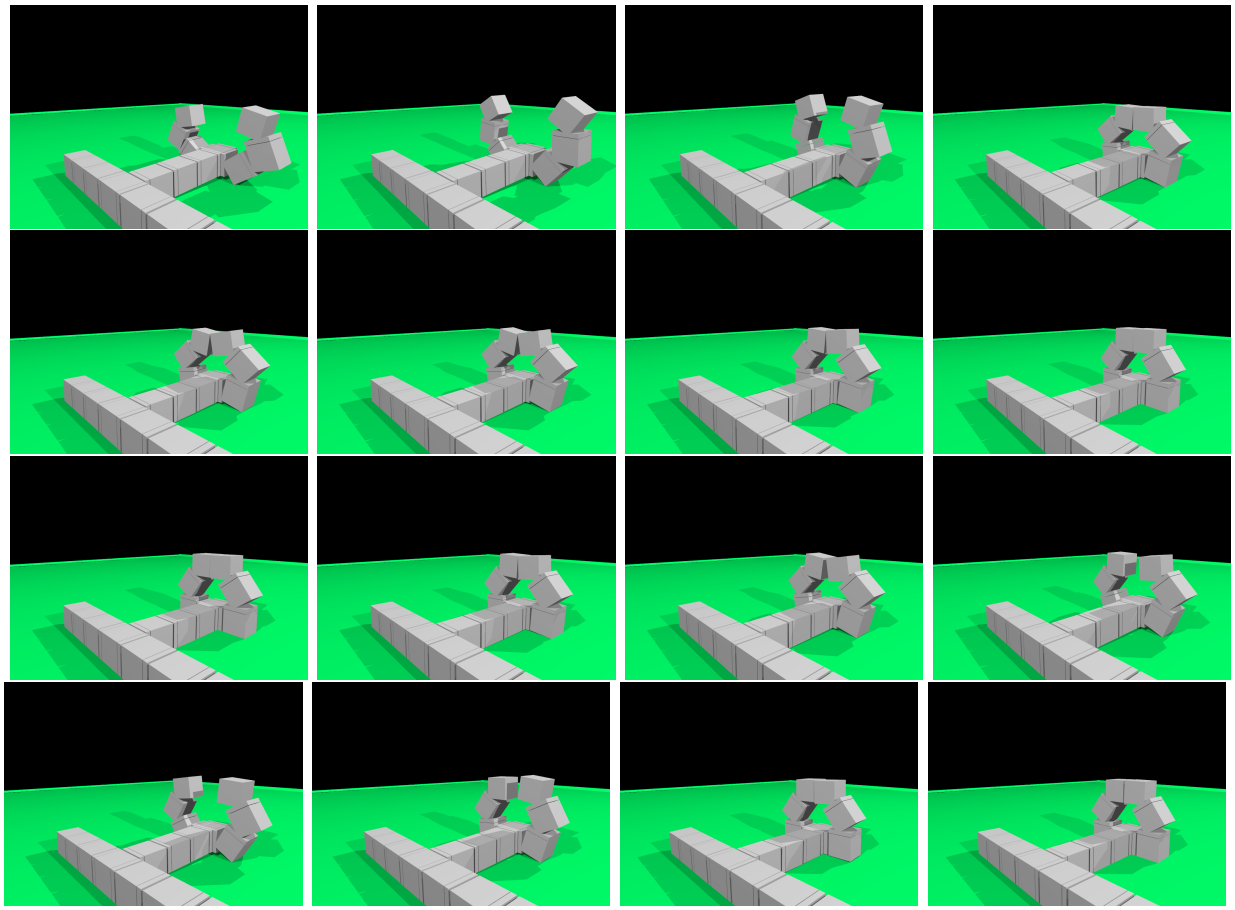

Figure 7.3: Movement of a robotic creature with two functional legs.

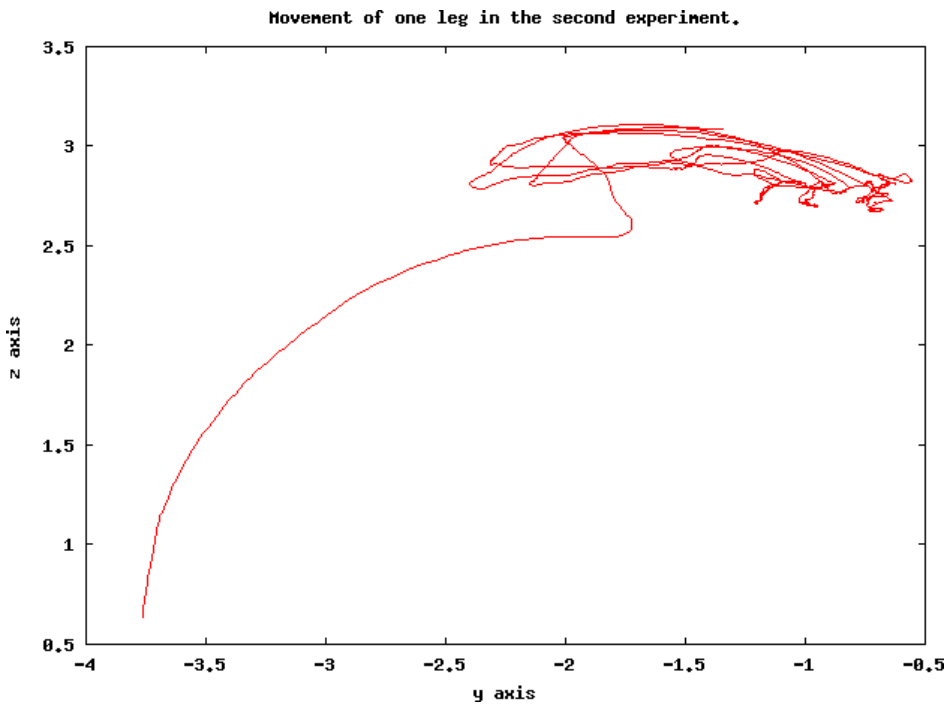

Figure 7.4: Trajectory of a single leg in the second experiment.

As the recorded trajectory represents the movements of the centre of the leg segment, it never actually passes trough 0 as would be expected, but it is clearly visible from video and Figure 7.3 that legs touch themselves by the whole sides of their last segment.

This experiment shows that synchronization of two legs woks as expected.

7.2.3 Four Legged Walking Robot

With the both EA-f and EA-s algorithms successfully tested, everything has been prepared for the main experiment. This experiment is actually repeated two times with two different robot configurations. Both of those were described in Section 2.3.

7.2.3.1 Sideways walking Robot

First is the robot designed to walk sideways like a crab. There were several trials of this experiment with different but still interesting results. First trial was done with basic setting without the focus on proper walking patterns with only goal, for the robot to move as far as possible. The video recording of the final creature can be found in video-file *Robot05.avi* and several frames in Figure 7.5.

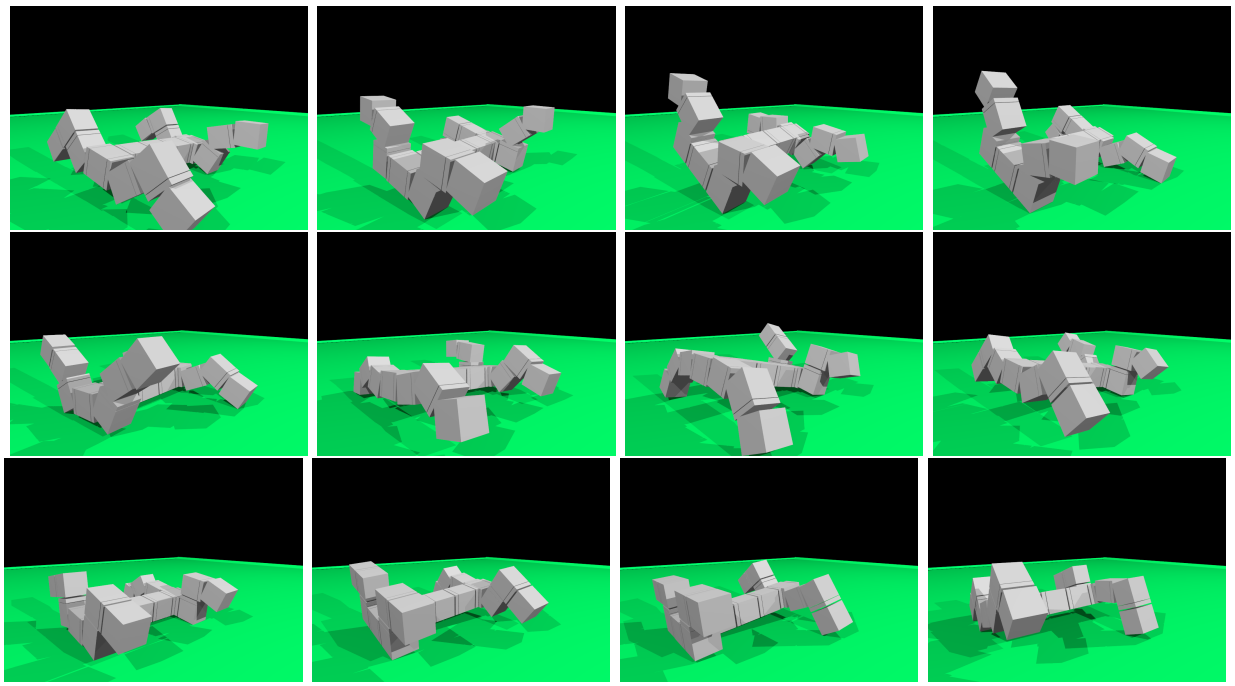

Figure 7.5: Movement of a robotic creature with all four legs enabled.

It is clearly visible that robots drags its body on the surface without any visible tendency to hold it upwards. As this is clearly not the desired behaviour the penalty to fitness was added as described in Section 7.1.3.1.

But when this penalty was set too high it lead to a different problem. One such robot can be seen in video *Robot06.avi* and in Figure 7.6. This creature easily hods its body up and does not fall over but has problems to walk in any one direction and just moves randomly near the starting point.

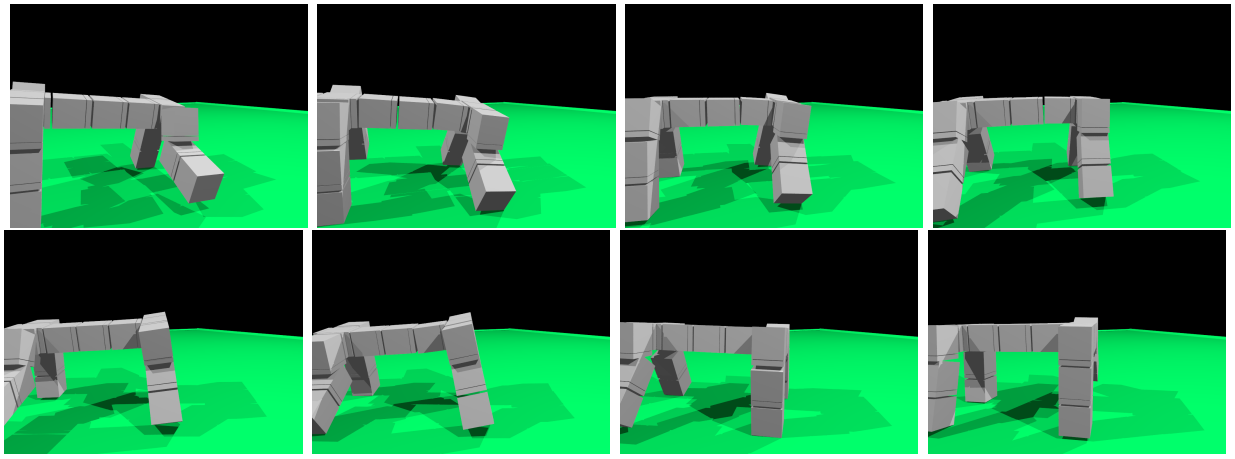

Figure 7.6: Movement of a robotic creature with four functional legs and too high body posture.

In the end it turned out that best results can be achieved with the parameter of body height penalization set to force the robots torso between 0.8 and 1.2 of the height of one block. Both boundary values led to different motion patterns but both can be described as success.

The robot with body held higher is recorded in video *Robot08.avi* and in Figure 7.9. The trajectory of the robot is recorded in Figure 7.7 (the magenta line) and the distance covered in Figure 7.8. This robot walks much faster than any previous one and also goes in quite a straight line although it has a tendency to turn left.

The robot with lower holding of the torso is in video *Robot07.avi* and in Figure 7.10. The trajectory of the robot is recorded in Figure 7.7 (the red line) and distance covered in Figure 7.8. The main difference against the previous robot is in the way it uses its legs. While the first robot held last segments of legs pointed downwards and made most ot the movement with segments closest to its body, this robot points last segments sideways and uses the middle leg segments much more. This technique gives it more stability and allows for greater speed. This is the fastest robotic creature when the speed is measured as a distance in single direction over time. It also shows the tendency to turn left, but much less then the previous robot.

7.2.3.2 Forward Walking Robot

This experiment is very similar to the previous one but it features slightly different robotic creatures and the target point is set in front of the arena not on side like before. It was expected that these robot would be capable of much faster motion and their learning would

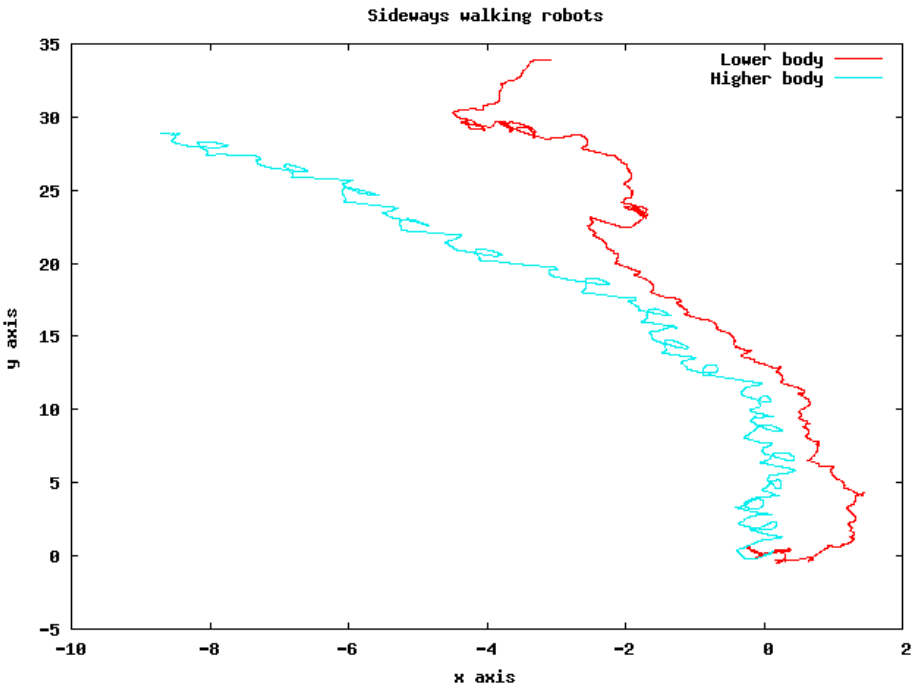

Figure 7.7: Trajectory of sideways walking robots.

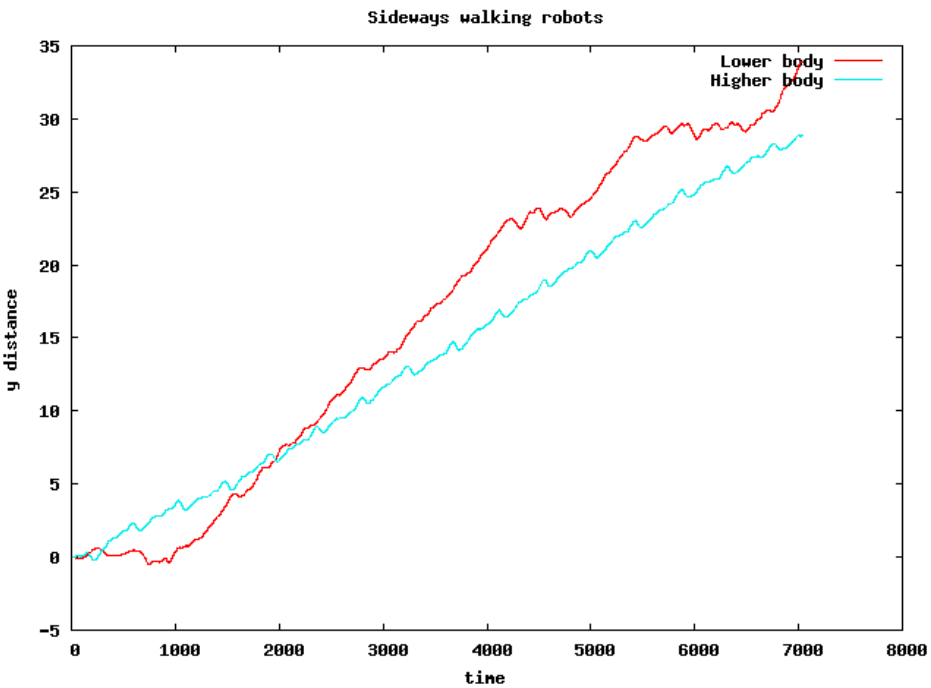

Figure 7.8: Distance covered by sideways walking robots.

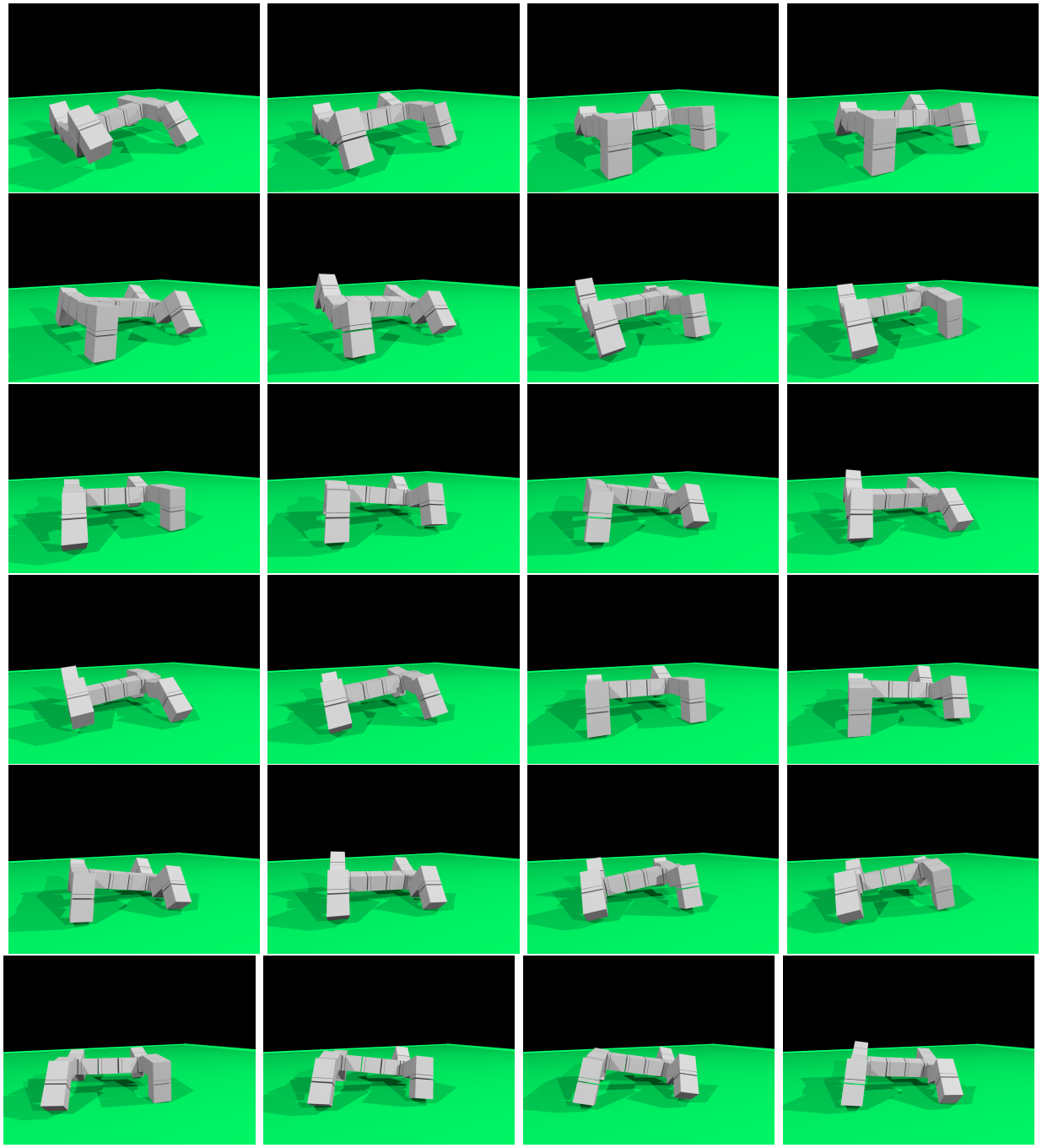

Figure 7.9: Movement of a robotic creature with higher body posture.

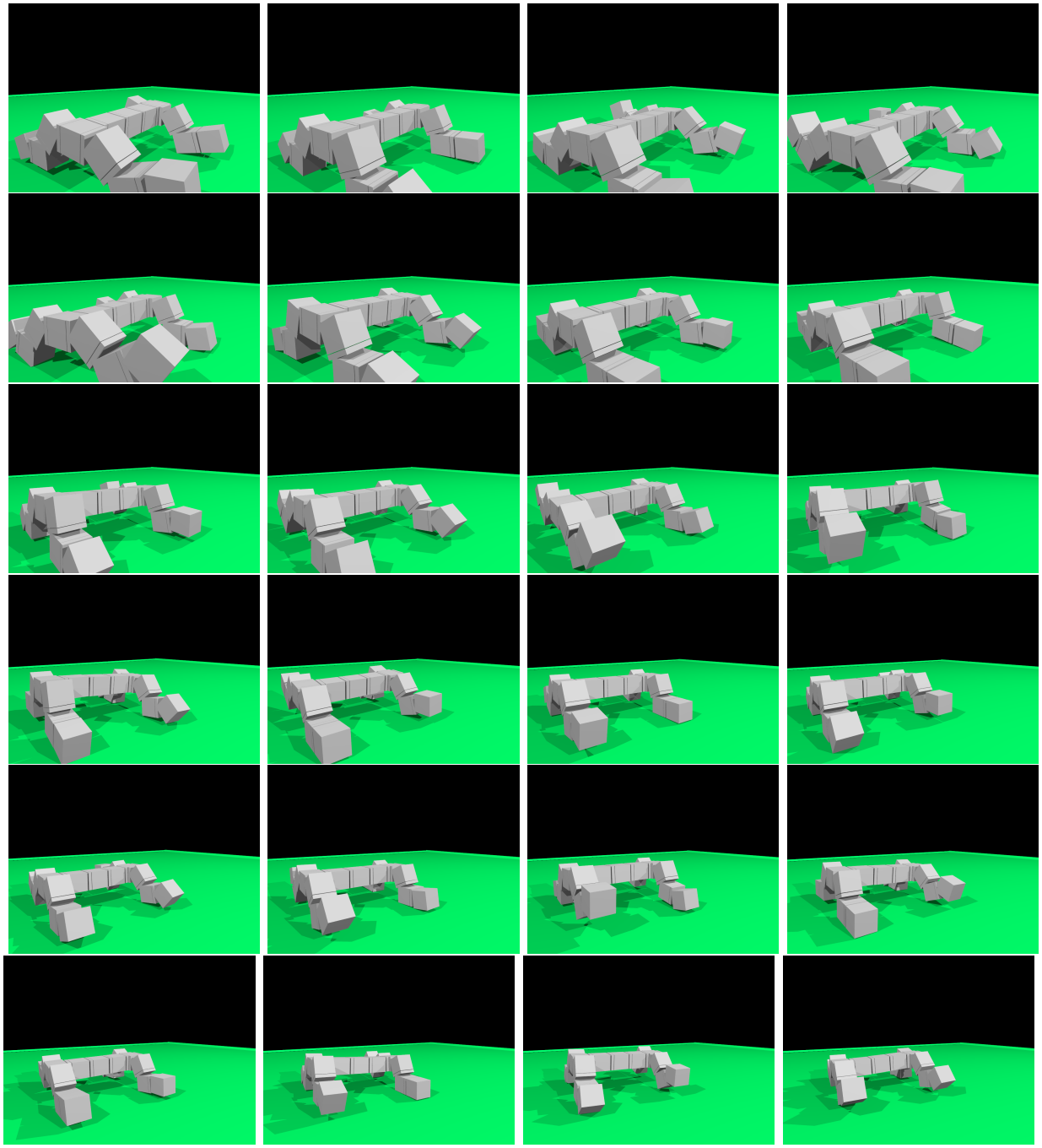

Figure 7.10: Movement of a robotic creature with low body posture.

be much easier. But they showed the same tendency to drag their bodies on the ground as the previous kind.

The problem was solved by the same technique and the resulting robot can be seen in video *Robot09.avi*, or in Figure 7.11.

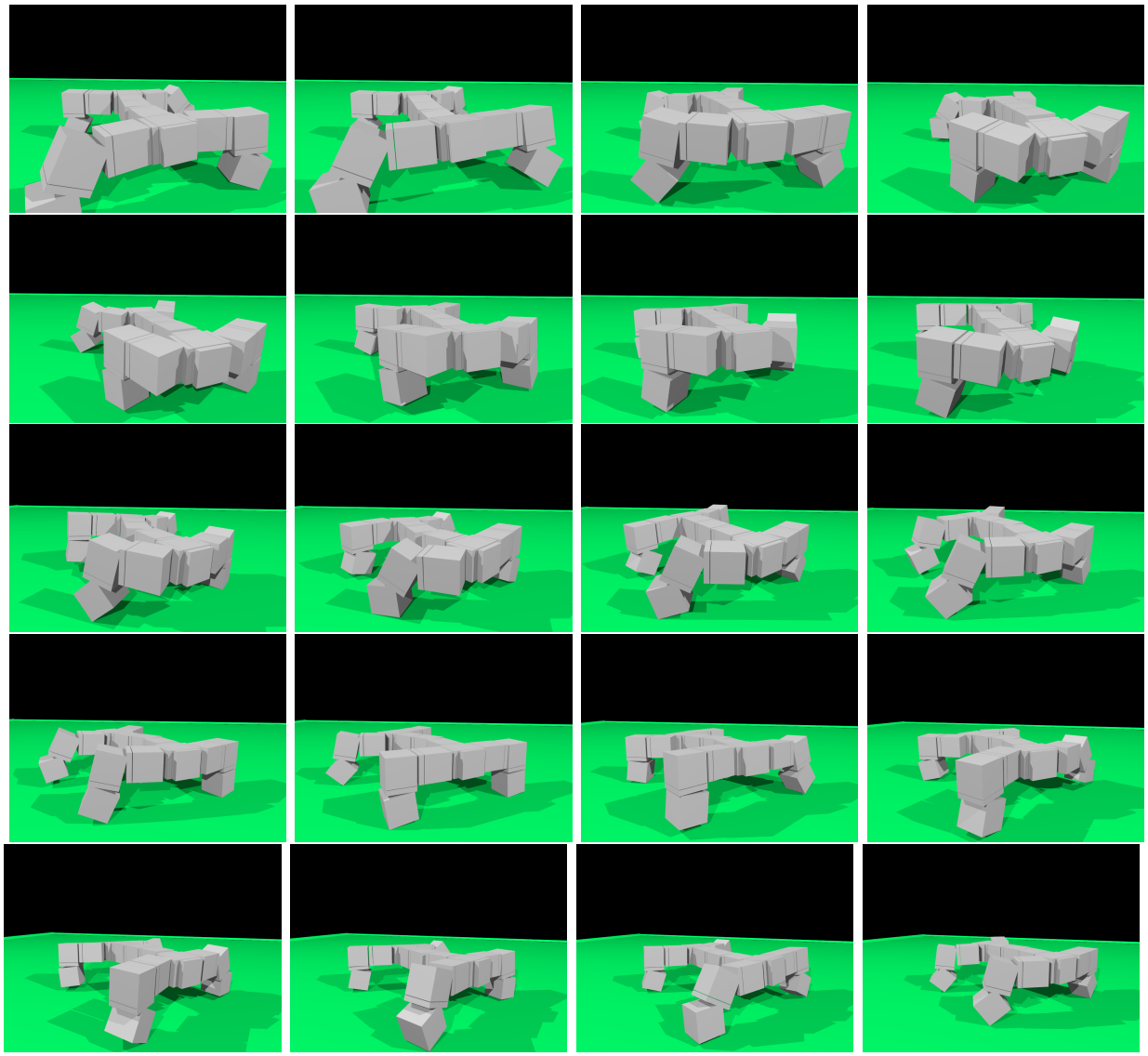

Figure 7.11: Movement of a forward going robotic creature.

However another problem showed up. This robot was unable to go straight and always walked in small circles as can be seen in Figure 7.13. This robotic creature is capable of the fastest movement of all but it has the worst result since it always returns to the starting point (see Figure 7.14 (red line) for covered distance).

The problem is that this creature got a rather high fitness during evolution. That is because the simulation which calculates the fitness runs for only 10 seconds and end before this creature turns around. To resolve this problem two method were used one at a time.

The first one was to use much longer simulation time during the evolution. The second was to penalize creatures that drift to much from the course.

Both those methods while different produced almost identical result which can be seen in Figure 7.12 and in video-file *Robot10.avi*.

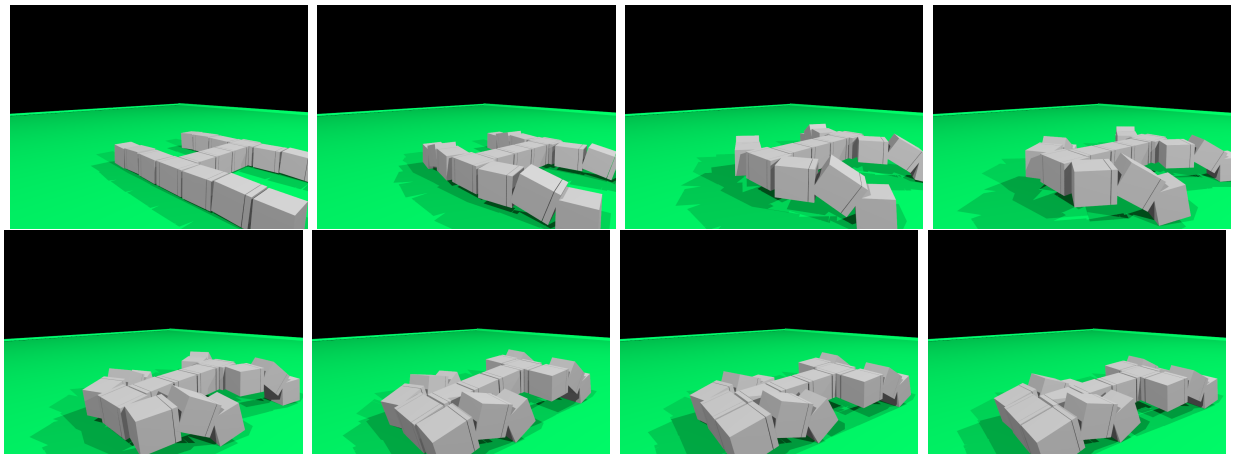

Figure 7.12: Movement of a Forward going robotic creature.

This robotic creature is the slowest from all walkers but it also goes in almost perfectly straight line. It also uses its legs in quite an interesting way. The other robotic creatures used mainly the leg segments closest to the body while this creature fixes those and the middle segments in one position and only uses the last segment to push itself forward.

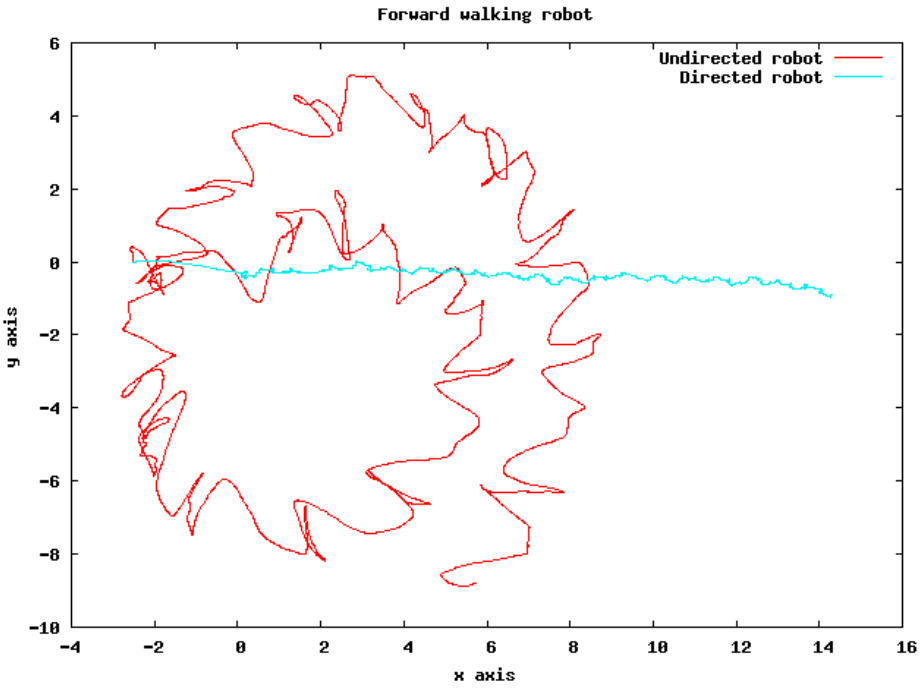

Figure 7.13: Trajectory of forward walking robots.

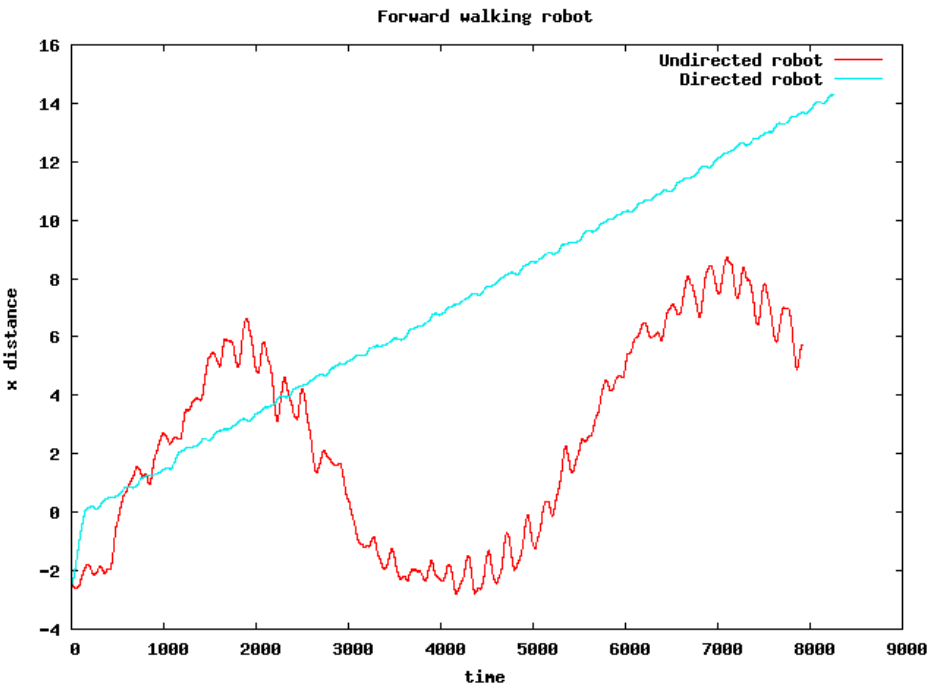

Figure 7.14: Forward distance covered by forward walking robots.

7.2.4 Synchronization Experiment

This is the last experiment as described in Section 7.1.4. While it would seem more logical to run this experiment among the first ones, this was impossible because control functions from the main experiment were needed.

This experiment was executed in two modifications. One with steady state evolutionary model and one with generational evolutionary model. Both algorithm used 40 individuals in population and were run for the same number of evaluations. The algorithm with the steady state model succeed in producing individual almost identical to that from which the control functions were taken, while the individuals generated by generational model turned out as significantly worse.

Chapter 8

Conclusion

The main objective of this work was to design a system capable of autonomously learning efficient motion patterns for the two predefined robotic creatures. In order to accomplish this task a co-evolution approach consisting of two related problems has been selected. The two proposed algorithms used in this co-evolution design were implemented and interconnected with the *Sim* robotic simulator.

The results accomplished with this approach are quite satisfactory. Two possible motion patterns were discovered for the first (crab-like) of the two robotic creatures used in the experiments. The main difference between those two is in the way the leg segments are used. In the first of these patterns the middle segments of the legs are fixed and only the first and last segments generate the movement while the second pattern employs first and middle segments of the legs much more. The second motion pattern is slightly better at maintaining direction. The first one is better at overcoming small obstacles like the walls of the arena. Both of these motion patterns allow for a quite fast movement with a small enough deviation from the forthright direction.

The results for the second (mammal-like) robotic creature are not as perfect as in the case of the first creature but are still interesting. After several experiments two possible motion patterns were discovered for this type of robot. The first one, while allowing for a really fast motion of the robot, was unable to keep it heading in forward direction. The second one was the other extreme. The robot moved in an almost perfectly straight line but at a very low speed. It may be possible that the results for the forward moving robotic creature might have been better if it had been designed with a "tale" section to balance the body the same way some real animals do it.

Robotic creatures of this size are not yet ordinarily used mainly because the needed hardware is still in development stages. However this thesis shows that the motion pattern for walking exists and with minimal modification of the fitness function other types of movements can be generated. Those atomic moves can later be used to navigate the robot in a more complex environment.

The modification of the robotic creatures is one of the tasks for further experimentation. Creatures with heads and tails, with six or even more legs or other interesting configurations can be examined and simulated. The ultimate future goal is generating motion patterns for creatures which themselves were automatically assembled. Moreover the Evolution Algorithm itself can be further developed. For example different representations of individuals in both EA populations can prove advantageous or the co-evolution scheme can be redesigned to allow for lower number of necessary evaluations.

Bibliography

[1] T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, USA, 1996.

[2] D. Beasley, RR Martin, and DR Bull. An overview of genetic algorithms: Part 1. fundamentals. *University computing*, 15:58–58, 1993.

[3] N.L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms*, volume 183, page 187, 1985.

[4] M. Fujita. Aibo: Toward the era of digital creatures. *The International Journal of Robotics Research*, 20(10):781–794, 2001.

[5] D.E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1:69–93, 1991.

[6] D.E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms (FOGA 1)*, volume 1, pages 69–93, San Francisco, CA, USA, 1991. Morgan Kaufmann.

[7] J.H. Holland. *Adaptation in natural and artificial systems.* Number 53. University of Michigan press, 1975.

[8] S. Kernbach. Awareness and self-awareness for multi-robot organisms. *Arxiv preprint arXiv:1111.5219*, 2011.

[9] S. Kernbach, E. Meister, F. Schlachter, K. Jebens, M. Szymanski, J. Liedke, D. Laneri, L. Winkler, T. Schmickl, R. Thenius, et al. Symbiotic robot organisms: Replicator and symbrion projects. In *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*, pages 62–69. ACM, 2008.

[10] N. Kohl and P. Stone. Machine learning for fast quadrupedal locomotion. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 611–616. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2004.

[11] J.R. Koza. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems.* Stanford University, Department of Computer Science, 1990.

[12] J.R. Koza. Genetic programming: On the programming of computers by means of natural selection, 1992. *See http://miriad. Iip6. fr/microbes Modeling Adaptive Multi-Agent Systems Inspired by Developmental Biology*, 229, 1992.

[13] S. Luke and L. Panait. A survey and comparison of tree generation algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 81–88. Citeseer, 2001.

[14] S. Luke and L. Panait. Fighting bloat with nonparametric parsimony pressure. *Parallel Problem Solving from Nature—PPSN VII*, pages 411–421, 2002.

[15] L. Panait and S. Luke. Alternative bloat control methods. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 630–641. Springer, 2004.

[16] Vojta Vonásek and Daniel Fišer. Sim: a general purpose 3d robotic simulator. May 2012.

[17] Jürgen Wakunda and Andreas Zell. Median-selection for parallel steady-state evolution strategies. In Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, Juan J. Merelo Guervós, and Hans-Paul Schwefel, editors, *PPSN*, volume 1917 of *Lecture Notes in Computer Science*, pages 405–414. Springer, 2000.

[18] Matthew Walker. Introduction to genetic programming. October 7 2001.

[19] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[20] D. Whitley et al. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the third international conference on Genetic algorithms*, volume 1, pages 116–121. Citeseer, 1989.

[21] R.P. Wiegand, W.C. Liles, and K.A. De Jong. An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, volume 2611, pages 1235–1245, 2001.

[22] A.H. Wright. Genetic algorithms for real parameter optimization. *Foundations of genetic algorithms*, 1:205–218, 1991.

Appendix A

List of Abbreviations

2D Two-Dimensional

3D Three-Dimensional

CX Crossover

DTM Distributed Task Manager

EA Evolutionary Algorithm

EAP Evolutionary Algorithms in Python

ES Evolutionary strategies

GA Genetic Algorithm

GP Genetic Programming

ODE Open Dynamics Engine

OSC Open Scene Graph

REPLICATOR Robotic Evolutionary Self-Programming and Self-Assembling Organisms

SSSA Scuola Superiore Sant'Anna

SUS Stochastic Universal Sampling

SYMBRION Symbiotic Evolutionary Robot Organisms

Appendix B

Content of Attached CD

```
Application
    Binary_Encoding                         Experiment with binary encoding
    Default_Implementation                  The main experiment
        evol
            fitness                         Evaluation code
                armController.cpp
                armController.hpp
                camera.patch
                fitness.pyx
                __init__.py
                Makefile
                robot.map
                robot_simulation.cpp
                robot_simulation.hpp
                sim
                sim.patch
                wood.ppm
            functions.py
            generic_population.py
            gp_population.py
            __init__.py
            param_population.py
            population.py
            tests                           Unit tests
                __init__.py
               test_fitness.py
                test_functions.py
                test_generic_population.py
                test_gp_population.py
                test_param_population.py
                test_population.py
        experiment.py                       Main experiment
```

```
        gnuplot.plt
        run_creature.py                     Creature display
    Secondary_Experiment
Python                                      Python installers
    Cython-0.16.tar.gz
    python-3.2.3.amd64.msi
    python-3.2.3-macosx10.3.dmg
    python-3.2.3-macosx10.6.dmg
    python-3.2.3.msi
    Python-3.2.3.tar.bz2
README
Text
    Cerny_2012.pdf
Video                                       Video-files with results
    Robot01.avi
    Robot02.avi
    Robot03.avi
    Robot04.avi
    Robot05.avi
    Robot06.avi
    Robot07.avi
    Robot08.avi
    Robot09.avi
    Robot10.avi
```