UNIVERSITY OF OSLO
Department of Informatics

Genetic Learning Algorithms combined with novel Binary Hill Climbing used for Online Walking-pattern Generation in legged Robots.

Master Thesis

Lena Mariann Garder

June 2006

Although Darwin was a clever man, he could not find out why only one ninth of his brain was working properly.....

Preface

This thesis is the result of a study for the Masters degree in Microelectronic Systems at the Department of Informatics, in the program for Electronics and Computer Technology at the University of Oslo. The thesis project was initiated in January, 2004 and concluded in June, 2006.

I am most grateful to my supervisor Mats Erling Høvin for his support, guidance and discussions, and for letting me experiment with his exceptionally fine robots. I would like to thank Dag Wisland and Jim Tørresen for their patience and fruitful discussions. I would specially like to thank Vidar Strønstad Øverås for supporting me, helping me and being able to cope with me through this period in my life.

Big thanks to the employees at the MES-group for making a great environment both socially and technically. Special thanks to Omid Mirmotahari, Karianne Øysted, Lene Hempel, Kjetil Meisal, Claus Limbodal, all of the guys at the VLSI-lab and the inhabitants of 'Lekestue'. In addition I thank my friends outside of the University for great social support.

I would like to thank my mother and father for raising me this way so I was able to survive this educational ordeal, my siblings, Anne, Bjørn and Kjetil, and the rest of my family for their loyal support in every way throughout my work. Last I would like to thank God Almighty for taking care of me, my life would not be the same without You.

Lena Mariann Garder
June 2006

Abstract

According to Darwin every species on this planet have developed from a small group of simple molecules into all the modern species living among us today. The reason why some species survive and others don't is what Darwin called Natural Selection, which means that every individual have to fight for its existence. Those who are best fit will survive. This has brought life to the well known saying: "Survival of the Fittest". The best fit will have the best chance to reproduce, to pass its well fitted, surviving qualities on to their offspring. And the offspring of two well-equipped parents will have a high probability of adaptation, and so the circle of life goes on...

A set of evolutionary search methods have been extracted from the Darwinian theories of evolution. These have been evolving in computer environments for several decades and have been passing through different areas of computer science, from theoretical tuning problems, algorithm developing, clustering, chip design, and several real world applications have been the foci the last years.

In this thesis Genetic Algorithms and Evolvable Hardware is used for evolving gaits in a walking biped robot controller. The focus is fast learning in a real-time environment. An incremental approach combining a genetic algorithm with hill climbing is proposed. This combination interacts in an efficient way to generate precise walking patterns in less than 15 generations. Our proposal is compared to various versions of Genetic Algorithms and stochastic search, and finally tested on a pneumatic biped walking robot.

Contents

Preface iii

Abstract v

1 Introduction 1
1.1 Introduction 1
1.2 Main Motivation of the Thesis 3
1.3 Important Headlights 5
1.4 Outline 5

2 Previous Work 7
2.1 Gait Development in Walking Machines 7
2.2 Real World Applications 10
2.3 Learning Algorithms in Digital and Analog Circuits 10
2.4 Robotic Remedies 12

3 The MES Robot, Hardware & Software 15
3.1 Hardware: The Robots 16
3.2 Software: The Simulator 17
3.3 The Line of Action 19

4 Genetic Algorithms 21
4.1 History of Genetic Algorithms 21
4.2 Genetic Algorithms 22
4.3 Chromosome Coding of the MES Robotic Chicken 23
4.4 Pauses 24
4.5 Stochastic Search 25
4.6 Stochastic Search on the MES Robot 26
4.7 Genetic Operators 27
4.8 Introducing GA to the MES Robots 32
4.9 Modified GA Runs 33
4.10 Various Selection Models: Ranking Selection 38
4.11 Various Selection Models: Tournament Selection 39
4.12 Various Selection Models: Evolutionary Strategie 41

4.13 Testing the ES Selection Model 43
4.14 Testing the ES Selection Model, Constant Pauses 44

5 The Incremental Approach **47**
5.1 The first Incremental Approach 47
5.2 Predefined Pauses in a Tree Structure 49
5.3 Incremental Results Using Pause Tree. 51
5.4 The GA - Binary Hill Climbing Algorithm 53

6 Measured Results **61**
6.1 Gaits Obtained 61
6.2 Practical Challenges 62
6.3 Measured Results 63

7 Conclusion and proposal for further work **65**
7.1 Conclusion 65
7.2 Discussion 67
7.3 Other Ideas 68
7.4 Idea: Mother Routine 68
7.5 Idea: Positive Reinforcement 68
7.6 Idea: Control Method / A priori Knowledge 69
7.7 Idea: Cyclic 69
7.8 Idea: Reinforced Learning 70
7.9 Further work 70

A GECCO 2006, Published Paper **71**

B Incremental Appendix **79**
B.1 Successful Incremental Approaches 79

C Media **85**
C.1 TV performances 85
C.2 Radio performances 85
C.3 Newspaper articles 85
C.4 Magazine articles 87
C.5 Invited Talk 87

D Dictionary **89**

E Source Code **93**

Bibliography **111**

Chapter 1

Introduction

1.1 Introduction

To find a solution to a given problem, there may be several ways to search. You are most likely to find one global optimum, being the best solution to your problem. There might however, exist several local optima, that are close enough and thus saving you a lot of searching, cost and effort. If your local optimum saves the purpose of being sufficient, it might substitute the global optimum.

The dimension or space, in which you are searching, can be of very different characters. The space might be flat as a floor in an empty room, or chaotic like a jungle. The different ways of searching will vary, depending on the landscape. When maneuvering in a chaotic search space, there are several methods for finding optima. An example of a chaotic search space may be learning how to walk. In this context all possible ways of moving is the search space, this is described by Hornby et. al. in their work with Aibo [1]. Another example of a chaotic search space is to find a system to explain the origin and the dynamics of how sun storms appear and move. Tracking rules in different languages is another way to describe a chaotic search space. Languages have often been cultivated through people's ways of expressing themselves through thousand of years, and serves as an example of a chaotic system.

This thesis focuses on Genetic Algorithms (GA) inspired by natural selection. The methods proposed in this master thesis can be applied to nearly any search space. The requirements are to be able to control the input, and receive a feedback. The feedback is a value that represents how well fitted the solution is to a given problem. For the GA approach to be profitable compared to other search algorithms the search space need to be chaotic, without any obvious logic. Otherwise other methods are found more suitable, e.g. Hill climbing [2] and Neural Networks [3] [4].

Figure 1.1: *Henriette, the MES robot chicken, learning how to walk.*

Developing effective gaits for bipedal robots is a difficult task which requires optimization of many parameters in a highly irregular, multidimensional space. In recent years biologically inspired computation methods, and particularly GAs, have been employed by several authors. For instance, Hornby and colleagues used GA to generate robust gaits on the Aibo quadruped robot [1]. Several authors have performed similar experiments, this is further described in Chapter 2. One of the main objections to the application of GAs in the evolution of gait is the one concerning the notable time-consuming characteristics of these techniques. In order to reduce the time spent to evolve a proper robot gait, various experiments for speeding up the GA have been executed.

To reduce the time spent several authors have separated different parts of their problems/search spaces from one another or they have incrementally divided the search space into subtasks. The incremental approach is further presented in Chapter 5. For an exhaustive description of several incremental approaches, readers may refer to the article written by Cantu Paz [5].

Figure 1.2: *The Asimo, from Honda, are pre-programmed to perform certain tasks when being introduced to it.*

1.2 Main Motivation of the Thesis

Robotic gaits can also be pro-programmed by a human. The humanoid robot Asimo is a successful example of a robot equipped with a database of pre-programmed gaits. Although Asimo is able to run, climb stairs and otherwise is very impressive it is far from comparable to natural beings with respect to elegance, efficiency, ability to adapt to environmental changes and robustness. On the contrary there is no pre-programming in evolution. The main motivation for this thesis is therefore to investigate if evolution can be used to develop gaits in an artificial being like the robot chicken "Henriette", illustrated in Figure 1.1, in real time.

The scientific background is based on the theory that most things, at least everything alive, has evolved by evolution [6]. The dinosaurs came and were annihilated probably because they lacked the ability to adjust properly to the environment. The modern humans adjusted through generations. They have grown taller and they live longer.

Another example is species of fish that live in the deep subterranean lakes without ever experiencing day light. There are even scientists that claim to have found signs of bacterial life on the moon called Europe, which circles around Jupiter. This moon has a lot of volcanic activity and is therefore highly toxic. The examples of plants, humans and animals adjusting to the environment and the everyday life are several and diverse.

Figure 1.3: *Intelligent design of higher powers or Darwinian evolution.*

The thesis is not meant to challenge neither religions nor natural science, but does not disregard the possibility that natural evolution is being "controlled" by a creating higher power. William Dembski [7] argues that nature has been intelligently designed and can not be explained solely by Darwinian evolution.

Many scientists and pioneers have concluded that the "survival of the fittest" is the basis of evolution. This theory was primarily proposed by Darwin [6]. When modern scientists apply this knowledge in form of computer languages, (i.e. evolutionary algorithms), they are surprised to find that their systems do not function properly. Drawing a parallel to modern evolutionary research it seems impossible to match the results given by the natural processes by simulations in a computer. So the question is what role does this creating higher power play in the evolutionary process?

This thesis sets out to solve the evolutionary challenge by optimizing algorithms, for this Genetic Evolutionary influenced chicken robot called Henriette. Illustrated in Figure 1.1

1.3 Important Headlights

The majority of scientists working within this field only experiment in software because of the great difficulties often connected to hardware evolution. But simulations do not necessarily correlate with the real world as we will see later in this thesis. A crucial requirement for real-time evolution is that it has to be fast as most mechanical robot parts will not last for millions of generations. The focus in this thesis has therefore been to optimize GA to enable fast real-time evolution and the tool has been incremental techniques with as little a priory knowledge as possible.

The work described in this thesis is divided in two parts - the first work is done on a search space formed by representing the gait chromosome with 4 bits pause lengths. The other work is done on a search space formed by a 6 bits pause representation (see paper in Appendix A). These two search spaces are different and therefore the results differ slightly.

- **Stochastic Search** has been tested on the robot as a comparative reference.
- **Simple GA** has been tested, performing considerably better than the stochastic approach.
- **Modified GA** has been tested improving the performance.
- **A novel tree-structure** incremental GA has been proposed and tested improving the performance further.
- **A novel incremental GA algorithm named GABH** has been proposed and tested providing the best results in this work.

1.4 Outline

- **Chapter 1** gives a light introduction to the thesis and its content, it also presents the phenomenon search space.
- **Chapter 2** contains background material, the scientific work that underlies this thesis, and other robots of current interest.
- **Chapter 3** introduces the MES robots and a description of their construction. There is also information about the simulator used in the work that underlies this thesis.
- **Chapter 4** describes the Genetic Algorithms, theirs history and functionalities. It also describes and illustrates the results of a stochastic search in the search space of evolving gaits of the MES robots. The chapter further gives a

thorough description of the genetic operators and demonstrates a first search with GA applied on the biped MES robot. Then modified GA search are presented and different selection models are described and tested. The large differences in fitness score when varying the pause lengths within the chromosomes are also discussed.

- **Chapter 5** contains a description of different incremental approaches performed by other researchers. This chapter describes several ideas tested to speed up the evolution of gaits on the MES robot. The results obtained by these methods are also presented. This chapter further contains the novel GABH algorithm applied on the MES biped robot.

- **Chapter 6** shows measurements of robotic gaits and other results.

- **Chapter 7** presents a conclusion, gives a brief overview of tested ideas, and gives a proposal for further work.

Chapter 2

Previous Work

This chapter includes the scientific background material that underlies the work done in this thesis. The emphasis is walking machines, and real world applications evolved by the use of learning algorithms.

2.1 Gait Development in Walking Machines

When the work in this thesis is described, the responses are often questions about similarities compared to AIBO. The funny SONY robot dog is an electronic pet, illustrated in figure 2.1, that most people know. The AIBO is pre-programmed to behave like a dog, with a lot of a priori knowledge included into the program controlling it. The AIBO even contain some of machine learning performing in cycles.

Figure 2.1: *The SONY robotic pet called Aibo.*

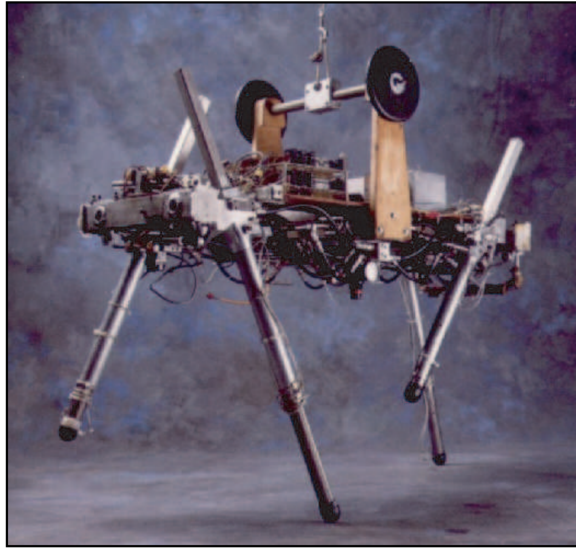

Figure 2.2: *The MIT quadruped robot.*

The gaits of the AIBO robot was evolved by Hornby et al. [1], they were evolved by Evolutionary Algorithms. The feedback was given by the cameras inside the eyes, measuring distances and giving feedback. In recent years, biologically inspired computation methods have been employed to find walking patterns by several authors. Ullerstam and Mizukawa [8] present a reprogrammable Aibo, which learns both the goals, sub goals and the means to achieve them during real-time interaction with humans. They conclude that complex behavior patterns can be learnt by a system based on Reinforcement Learning (RL) [9]. RL is an evolutionary method inspired by animal learning. The work done with the hexapod robot named *Kafka* [10] is worth mentioning. This robot evolves gaits by the use of Q-learning [11], which is applied directly on the robot hardware. Q-learning is a form of RL. RL was also tested online on the *Micro Electronics System Group* (MES) robot Henriette, but the results were not advantageous because of the practical challenges mentioned in Chapter 3.

Gait evolution on hexapod robot by the use of neural network is presented by Lewis, Fagg and Bekey [12]. They use GA to control the neural network, and conclude that it is impractical to apply GA in a strait forward manner to the design of a neural network to control a robot. But the approach introduced by Lewis et. al called "island of converges", makes the use of GAs practical by applying GAs to real robots.

The Massachusetts Institute of Technology (MIT) have earlier been one of the leading within robotics combined with learning algorithms [13]. The Leg-lab was founded in 1980 by Marc Raibert. One of the MIT Leg-Lab robots is pictured in figure

Figure 2.3: *The most advanced quadruped robot on earth, "Big Dog" from Boston Dynamics.*

2.2. When Raibert quit from MIT Leg-Lab in 1995 he co-founded Boston Dynamics [14] which now has, according to themselves, the most advanced quadruped robot on earth, called Big Dog, see figure 2.3.

GA applied to bipedal locomotion was also proposed by Arakawa and Fukuda [15] who made a GA based on energy optimization in order to generate a natural, human-like bipedal gait. Energy optimization is just one of the techniques tested to reduce the time spent searching in large search spaces with evolutionary methods like GA. One of the main objections to applying GAs in the search for gaits is the time-consuming characteristic of these techniques due to the large fitness search space that is normally present. For this reason most approaches have been based on offline and simulator based searches. Various techniques for speeding up the algorithm will be presented as follows. The increased complexity evolution scheme, introduced by Tørresen [16] has shown how to increase the search speed by using a *Divide and Conquer approach*, by dividing the problem into subtasks in a character recognition system. Haddow and Tufte have also done experiments with reducing the genotype representation [17]. Kalganova [18] has shown how to increase the search speed by evolving incrementally and bidirectional to achieve an overall complex behavior both for the complex system to the sub-system and from sub-system to the complex system. As earlier mentioned Cantu-Paz [5] further describes other approaches.

Figure 2.4: *Robot following a red ball (arranged picture).*

2.2 Real World Applications

Evolutionary algorithms have often been proposed as a method for designing systems for real world applications. Higutchi et al. [19] set a standard to the terminology within this field, and also mention several applications. Some of the real world applications are pattern recognition with GA in a prosthetic hand controller that makes it possible to train the hand prosthesis faster and therefore the user adapts faster to the prosthetic hand. They also present an adaptive robot controller, where a robot follows a red ball and avoid obstacles, illustrated in figure 2.4. GA controls the robot through sensory input. The results are 2 times faster in motion changes than earlier robotic controllers. Higutchi et al. also evolves different chips for cellular phones and describes decompression of an electro photographic printer.

The ball-chasing robot is thoroughly described by Higutchi [19]. Additional reading is presented by Higutchi and Yao [20]. Jim Tørresen [21] describes an Evolutionary approach to the control of a Prosthetic Hand, illustrated in Figure 2.5.

2.3 Learning Algorithms in Digital and Analog Circuits

The principles of evolutionary design are also used for designing digital circuits. Miller, Job and Vassilev [22] are pioneers in their work to explore new and better digital combinational circuits. Their main goal was to detect principles in smaller circuits that can be re-used for designing larger circuits in a more efficient way than conventional design methods. The evolutionary method used in their work is an Evolutionary Strategie (ES) called (1+λ) ES [23]. The ES will be further presented later in this thesis. Miller et. al concluded that the best result was obtained with limited inputs. The number of input in their 3 bit multiplier required 3 million generations. Even so the results obtained were uplifting as their evolutionary design

Figure 2.5: *Arm prosthesis (arranged picture).*

used less logic gates than the conventional circuits. They found it hard to extract principles of the evolved circuits. An arranged picture of a Xilinx-chip where evolution can be applied is illustrated in Figure 2.6

Koza, Bennet, Andre and Keane [24] have tried out the art of Genetic Programming (GP) [25] for automatic design of analog electrical circuits. The evolutionary method consists of three structures, where genetic operators like mutation, crossover, reproduction and fitness measure are applied. Their results where some years ago path-breaking, they rediscovered a conventional design for low pass filters. The solutions found by the GP, are of high standard that usually requires high human intelligence.

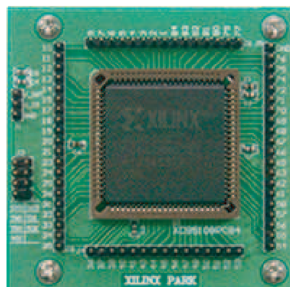

Figure 2.6: *Xilinx chip where evolutional methods can be applied.*

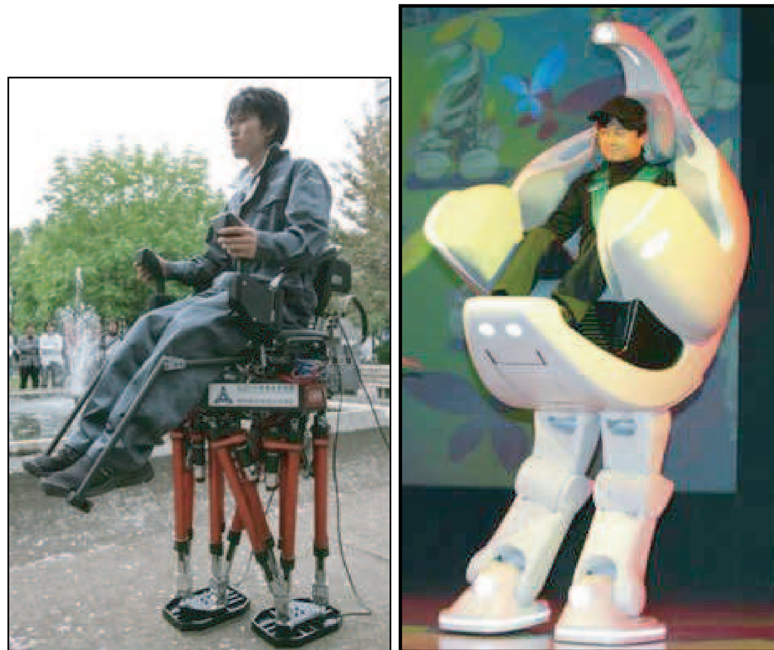

Figure 2.7: *Left picture, Waseda University in Tokyo, Japan, have made a walking robot-chair, applicable where wheelchairs don't reach. Right picture shows Toyota's "I-foot", controlled by a joystick inside.*

2.4 Robotic Remedies

There is a great research focus on developing remedies for handicapped people, this is not a part of this thesis. The photos in Figure 2.7 show that in a few years the wheelchairs may be obsolete. The latest news until summer 2005 where displayed to the public at the 2005 World Expo in Aichi, Japan.

Figure 2.8: *Strap-on robotic pants, intended to help soldiers or fire fighters carry heavy loads for long distances.*

The photos in Figure 2.8 show robotic help-legs called "BLEEX". This stands for "The Berkeley Lower Extremities Exoskeleton". It is a pair of strap-on robotic legs designed to turn an ordinary human into a super strider. Ultimately intended to help people like soldiers or fire fighters carry heavy loads for long distances, these boots are made for marching.
"The design of this exoskeleton really benefits from human intellect and the strength of the machine," says Homayoon Kazerooni, who directs the Robotics and Human Engineering Laboratory at the University of California-Berkeley.

Chapter 3

The MES Robot, Hardware & Software

The Microelectronic Systems Group (MES) robotic laboratory [26] consists of many different types of robots and walking machines. All the robots are handmade by Mats Høvin. The experiments have mainly been focused on a one-legged pneumatic construction called *Mono* (illustrated in Figure 3.1) and a two-legged pneumatic chicken robot called *Henriette* (illustrated in Figure 3.1). The four-legged robotic dog called *Turbo* (illustrated in Figure 3.2) and the robot-raptor called *JernÆrna* (illustrated in Figure 3.2) have only been preliminary tested. This chapter describes the facts concerning the construction of *Henriette* and the hardware information. The simulator is also described.

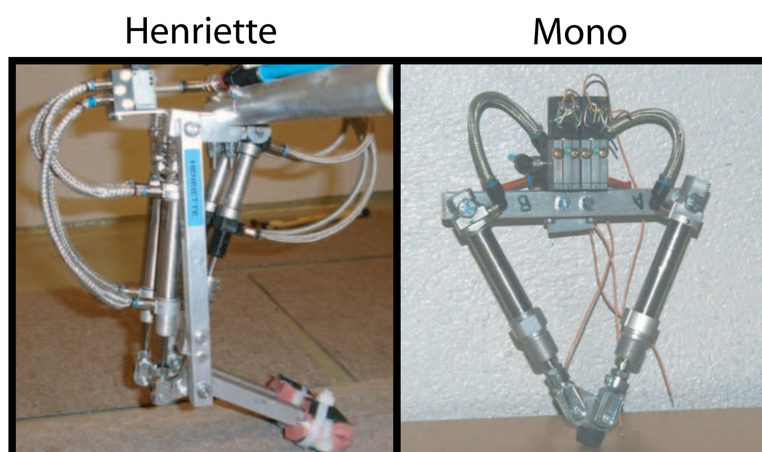

Figure 3.1: *The MES robots*

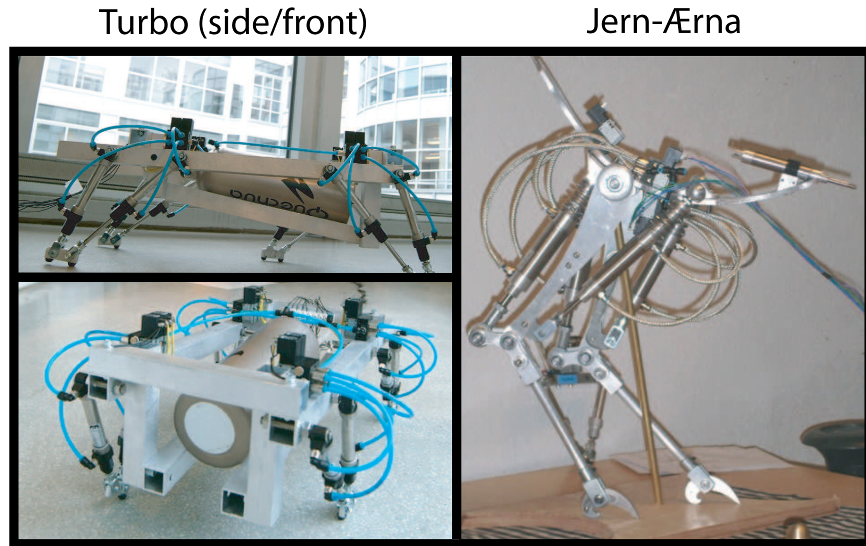

Figure 3.2: *The MES robots.*

3.1 Hardware: The Robots

The following description fits the robot called *Henriette*, a robotic chicken that is the object of most of the advanced experiments. The robot skeleton is made of light weight aluminum and is provided with two identical legs. The height is 40 cm. Each leg consist of an upper part (i.e. the thigh) connected through a cylindrical joint to the lower part (i.e. the calf). Pneumatic cylinders are attached to the thigh and the calf used for controlling the movements of the calf and the thigh separately.

As shown in Figure 3.3 the rear cylinder in each foot actuates the calf whereas the front cylinder actuates the thigh. The cylinders can either be fully compressed or fully extended, thus a binary operation. The pneumatic valves are located on top of the robot. The valves are electrically controlled by 4 power switches connected to a PC I/O card illustrated in Figure 3.4 element no. 3 (National Instruments DAQ-pad) and the different search algorithms are implemented in the programming language C. The pneumatic air pressure was set to 8 bar and provided by a stationary compressor. The robot was attached to a balancing rod at the top (Figure 3.3 right picture, and Figure 3.4 element no. 1) making the robot able to move

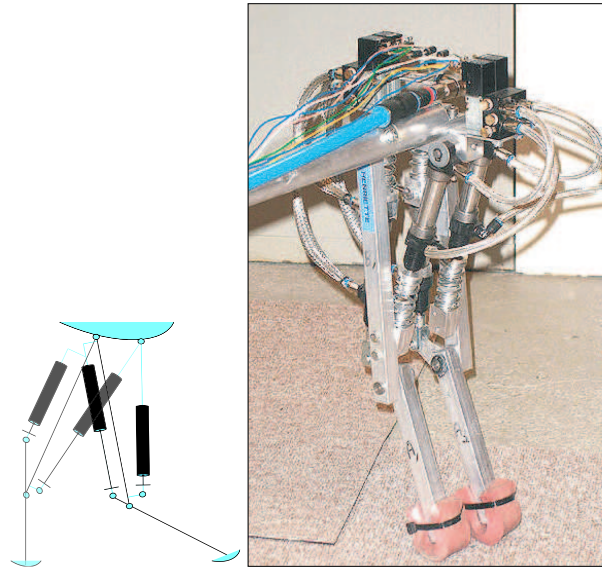

Figure 3.3: *An illustration to the left and a photo of the robot to the right. Proper walking direction is left to right (bird construction).*

in two dimensions. The other end of the rod is attached to a rotating clamp on a hub, illustrated in Figure 3.4 element no. 2. The robot walks around the hub with a radius of 2 meter, pictured in Figure 3.6

In addition to being a balancing aid, the rod supplies the robot with air pressure and control signals from the DAQ-pad. The hub has a built in optical sensor representing the rod angle in 13 bit gray code [27]. This is the feedback to the GA running on the computer. This optical angle measurer can be reset at any time, the GA is therefore provided with an exact measure of every movement, illustrated in Figure 3.5.

3.2 Software: The Simulator

A simple mechanical chicken-robot simulator has been implemented in the modelling language is C++. The simulator models the robot with exact physical dimensions and a weight of 3kg. The centre of gravity is located at the hip joint. It is found very difficult to model the feet-to-floor friction force exactly as this force is heavily modulated by large vibrations in the robot body and supporting rod during walking/jumping. The feet-to-floor friction force is a very important factor for developing efficient jumping patterns, and the lack of an exact model for this effect is assumed to be the main weakness of the simulator. The source code for the simulator is included in the appendix. To indicate the vibrations in the balance rod, 5%

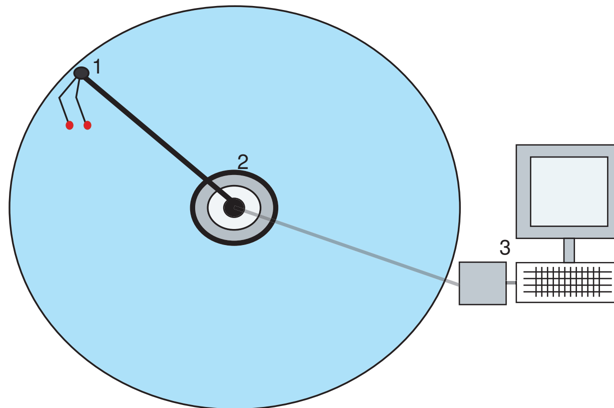

Figure 3.4: *The fitness measurement and balancing rod system, top view. The system consist of three elements, element 1 being the robot, element 2 being the balancing rod, element 3 the computer attached to the DAQ-pad.*

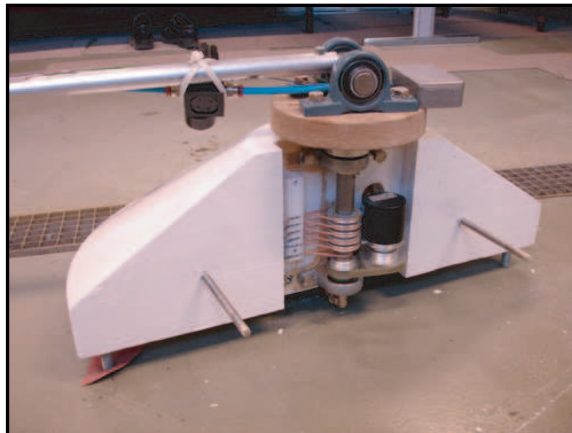

Figure 3.5: *Inside the hub. The black box is the 13bit optical/laser angle sensor. The other parts are control signal transmission and air (8bar) transmission.*

noise was added to the fitness landscape while running simulations. There are real time deviations in the computer running the XP operating system. The variations are measured to be up to +/- 15 ms.

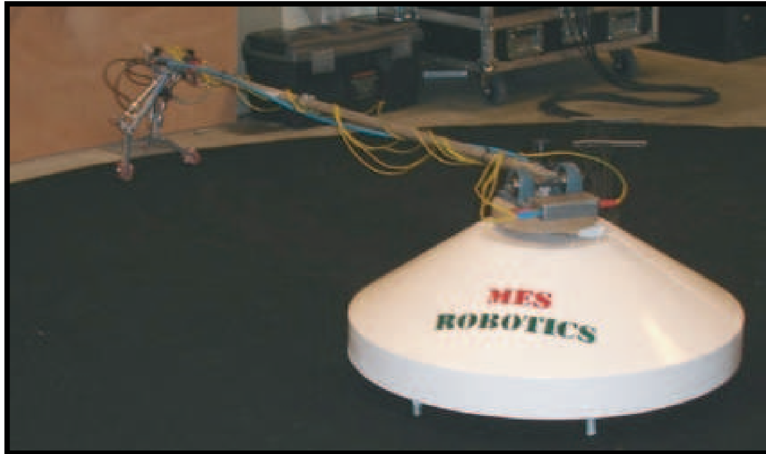

Figure 3.6: *The Robot, the balance rod and the hub.*

3.3 The Line of Action

The first approach was to exclusively run GA on the robot hardware, where evolution could perform without simulating the results in advance. The majority of experiments with GA are only tested in simulations. This master thesis was supposed to be the exact opposite. Three quarters of a year, diluted by additional master classes, were spent only testing the GAs on the robot called *Mono*, illustrated in Figure 3.1.

When evolving gaits on the robot, a lot of challenges arose. These challenges are thoroughly described in chapter 6. The tear and wear on the robot forced a simulator approach to the robot. By first simulating a lot of time was saved. Various simulations were undertaken to find the appropriate GA parameters. The ones that seemed to give the best results were selected and fixed for the robot experiments.

Chapter 4

Genetic Algorithms

4.1 History of Genetic Algorithms

Genetic Algorithms (GA) [28] originated from the studies of cellular automata, conducted by John Holland at the University of Michigan. Holland's book [29], published in 1975, is generally acknowledged as the beginning of the research of GAs. Until the early 1980s, the research in GAs was mainly theoretical [30], with few real world applications. This period is marked by ample work with fixed length binary representation in function optimization by, among others, De Jong and Hollstien. Hollstien's work provides a careful and detailed analysis of the effect that different selection and reproduction strategies have on the performance of a GA. De Jong's work focused on the adaptive features in the family of GA's that constitute a robust search procedure.

From the early 1980s the community of GAs has experienced plenty of applications spread across a large range of disciplines [19]. Each and every additional application has given a new perspective to the theory. Furthermore, in the process of improving performance, new and important findings regarding the generality, robustness and applicability of GAs became available. Following the last couple of years of intense development of GAs in the science, engineering and the business world, these algorithms in various shapes and forms have now been successfully applied to several problems. E.g. optimization problems scheduling, data fitting and clustering, trend spotting and path finding [31]. NASA have further applied GA in the design of an electromagnetic antenna [32]. Parts of the engine of the airplane Boeing 777 is also designed by the use of GAs [33].

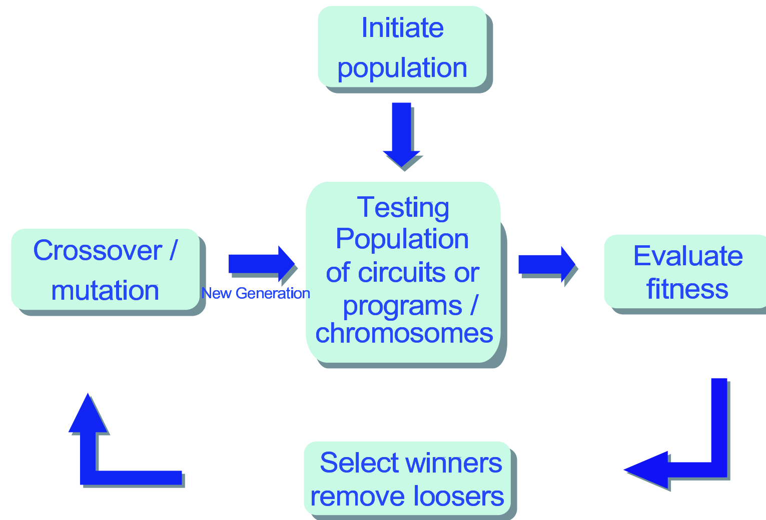

Figure 4.1: *An evolutionary System.*

4.2 Genetic Algorithms

The GA is a set of evolutionary algorithms that use principles of Darwin's theory of evolution [6]. It is not validated whether Darwin's principle alone is sufficient for successful evolution, but the GA is known to be useful for solving problems without deterministic algorithms. Since there are not algorithms known that deterministically develop robot gaits, GAs with some variations are chosen to improve the performance. In the GA simple approach, described by Goldberg [28] and by Tørresen [34], the idea is to represent a solution by a bit string, which is randomly initiated. The chromosome is also called an individual. There are several chromosomes in a population. The entire population is tested in the search space and the feedback (i.e. fitness) is retrieved. Then the genetic operators crossover, Figure 4.8, mutation, Figure 4.9, and reproduction (i.e. selection) are performed on the population and a new generation is tested toward the same problem. The operators and the representation are thoroughly described in the following sections.

A schematic illustration of an evolutionary system is shown in Figure 4.1. The population is situated on top of the figure, as it is randomly initiated. These chromosomes can also be seen as solutions that are tested towards a problem, this is what happens in the box in the middle of the figure. In the box to the right the fitness is evaluated, all the chromosomes are given a value that indicates how well the solution rated towards the given problem. The lower box represents the selection of parent-chromosomes to the next generation. In the next sections the different selection methods are described. The last events are the genetic operators crossover

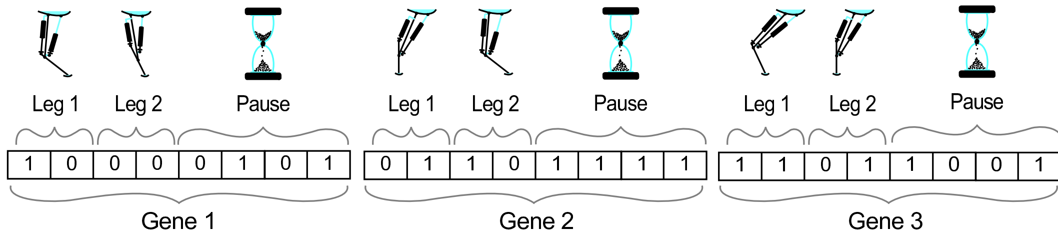

Figure 4.2: *Gait representation in a chromosome / bit-string.*

and mutation, seen in the box to the left. The crossover operator is the actual reproduction, like in nature the parent chromosomes are mixed together forming new chromosomes. The mutation is to add variation. The population has now passed through an entire generation, and again it is time to evaluate how well the new population is solving the problem, in the center box in Figure 4.1.

Summing Up the GA Procedure

1. Represent solutions by bit strings/chromosomes.
2. Initiate a population of solutions/chromosomes.
3. Test every chromosome and give a score of how well fitted the solution is/a fitness score.
4. Select desired amount of chromosomes, copy them into a new population.
5. Perform the genetic operators: mutation and crossover.
6. Go back to number 3.

4.3 Chromosome Coding of the MES Robotic Chicken

In our experiments each gait is coded by a 24 bit chromosome. The chromosome represents three body positions, each followed by a time delay (i.e. pause) of variable length. A body position is composed by the positions of the 2 legs (4 cylinders) and represented by four bits each, describing the status of the corresponding cylinder (compressed or extracted), as illustrated in Figure 4.2 and Figure 4.3. A complete gait is then created by executing 3 body positions with 3 appropriate pauses in between, illustrated in Figure 4.4. Each pause length is represented by 4 bits. The pause length is represented as a binary code corresponding to pauses from 50 ms to 300 ms. Two cylinders can move a single leg to 4 different positions, see Figure 4.3. Two legs with four cylinders can hold 16 different positions, and three following positions with 4 bits of pauses in between, make a search space of 24 bits, giving

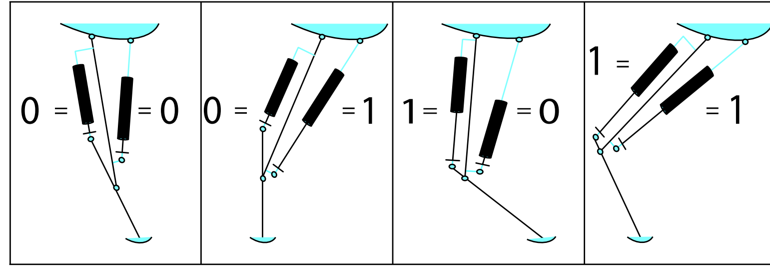

Figure 4.3: *Different positions for one leg of the MES robotic chicken.*

16'777'216 different walking patterns (i.e. gaits), see equation 4.1. Various simulations have shown no GA search speed improvement by representing the pauses in gray code.

$$2^{24bit} = 16777216 \tag{4.1}$$

4.4 Pauses

A gait is composed of leg positions and pauses. In our robot evolution we have found that the most efficient gaits with respect to forward speed are gaits dominated by jumping movements. In a jumping movement the pause length between each leg kick is outmost critical as the robot may stumble if the timing of the leg kick is just slightly wrong. Simulations show that a pause length deviation in the magnitude of 10 ms can make the difference between a relatively useless and a highly effective gait. It is however a trade off between the desire to represent the pause lengths with a high number of bits and the exponential decrease in search speed for each extra bit used due to the increased size of the search space. The number of pause bits representing the pauses in this thesis is 4. In the paper found in Appendix A, however, the number of bits representing the pauses is 6.

A plot of the chaotic fitness landscape is found in Figure 4.5, left plot, where the fitness is proportional to the speed of the robot. In this plot the different chromosomes are plotted after one another in a queue with their fitness value represented in the y-axis, making an irregular surface. The positions are variable, but the pause lengths are set fixed lasting 181 ms. The chromosomes are sorted by their gray value to keep bit changes as few as possible, but even so the landscape is chaotic

with many narrow peaks. The illustration is an example of the search space referenced from the paper, Appendix A.

In Figure 4.5 right plot, the search space of the pauses is plotted in three dimensions. Two of the dimensions are represented by the two first pauses, the third pause is fixed at 70 ms, and the third dimension is the fitness value found by simulating the entire pause landscape. The cylinder positions are kept constant. The pause-search space is an example referenced from the paper, Appendix A.

4.5 Stochastic Search

Stochastic search can be seen as a totally random selection, like the selection of numbers in a lottery. A stochastic search is a search method most would choose

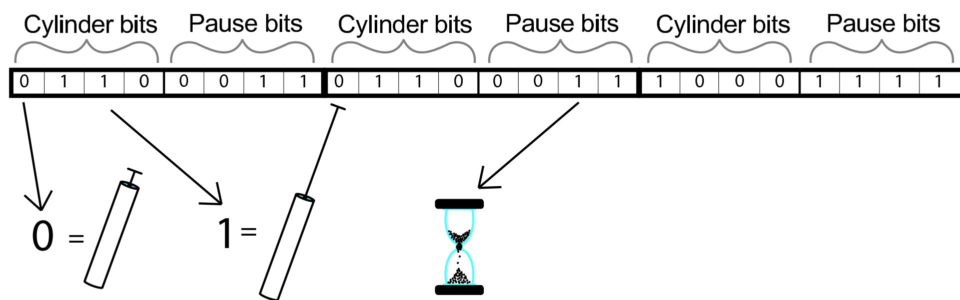

Figure 4.4: *Chromosome coding.*

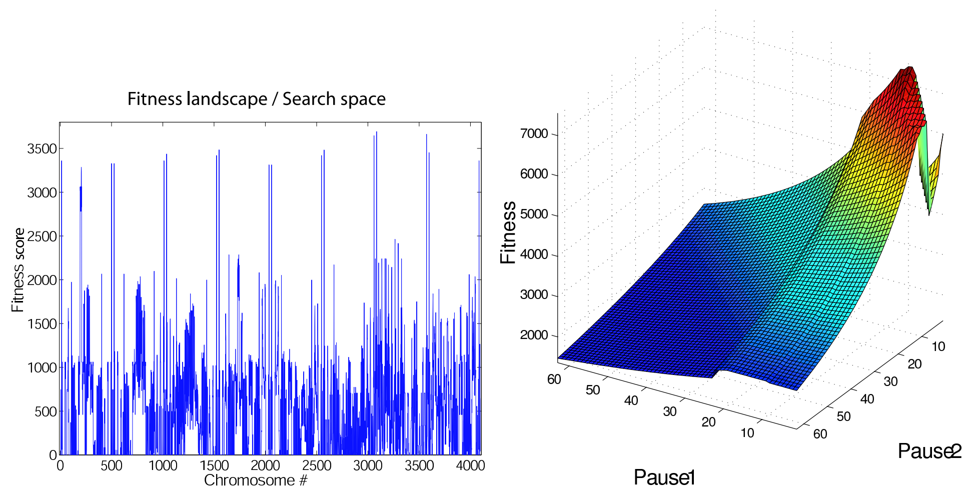

Figure 4.5: *The left plot illustrates the fitness search space for variable positions (fixed pauses at 181 ms) plotted after each other in a queue. The right plot shows fitness search space for variable pauses with constant positions.*

when being confronted with an illogical problem. It is a random selection of chromosomes (i.e. solutions) from all the possible chromosomes, measuring 16'777'216, see equation 4.1, if the pauses between the positions in an individual are variable. If they are constant there are 4096 different individuals, see equation 4.2

$$2^{12bit} = 4096 \tag{4.2}$$

The stochastically chosen chromosomes are tested towards the problem. If the tested chromosome scores higher than the prior ones, the chromosome is preserved. If the fitness score is lower than the prior preserved chromosomes, it is rejected. Stochastic search is testing random solutions until you are satisfied with the obtained solution.

4.6 Stochastic Search on the MES Robot

The stochastic search method is tested on the biped MES robot, *Henriette*. The results of the search are shown in Figure 4.6. The x-axis of the plot shows the time course denoted in generations. The y-axis shows how well the solution scored denoted in fitness scores. The fitness score is measured in $speed = angle/time = angle_of_velocity$. The angle of velocity is called fitness score.

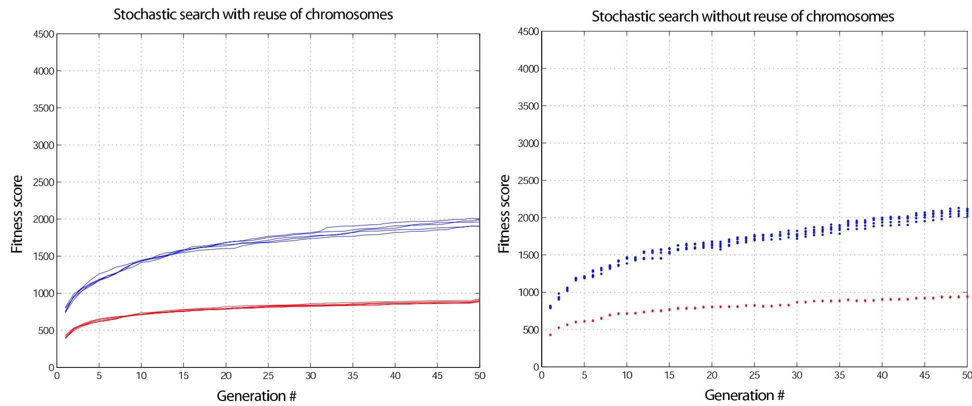

Figure 4.6: *Left plot shows Stochastic Search without reuse of chromosomes. The right plot shows Stochastic Search with reuse of chromosomes, shown in dotted lines. The y-axis represents the fitness score, the x-axis represents the development of generations. Each population contains 10 chromosomes.*

Figure 4.6 shows a plot of the results when stochastic search method was introduced to the problem of finding robotic gaits. The figure shows the max fitness in blue lines, and the mean fitness in red lines. The highest fitness score in the first generation measured a fitness score of 754. The mean of all the solutions presented

in that generation, both rejected and preserved, was at a score of 443. In the last generation, generation number 50, the best fitness measure is 2015 and the mean measure at this point is 967. These measures show that in 49 generations the population was able to increase the fitness measure by 63 %.
The dotted lines and the solid lines are quite similar to each other. Because of the large amount of chromosomes, there are 16’777’216 as illustrated in equation 4.1, while each of the lines in the plot only test 50’000. There is only a small possibility that there will be reuse of chromosomes. The plots show no significant difference of the two conditions, the reuse and the non-reuse, of chromosomes.
All the plots in this thesis include a max and a mean measure of the same runs. These mean and max graphs will for the future have the same color on the lines representing the same criteria for the search. They are, however, randomly initiated, so every chromosome will be different from one another in the first generation. One line does not represent one run of search, one line represent an arithmetic mean of 100 runs of search. This is done because there are large individual variations from one run to another. All the mean and the max values are represented by 5 lines each. There are 5 lines of each 100 runs to ensure accuracy in the plot since there might be a lot of disturbing factors while measuring fitness. This means that all the runs have 5 lines representing 500 runs of GA with the same criteria.

4.7 Genetic Operators

A proper presentation of the Genetic Algorithm-operators is necessary to understand the parameter tuning and the depth of the thesis. There are three main operators according to Goldberg [35], Crossover (Figure 4.8), Mutation (Figure 4.9) and Reproduction (i.e. Selection). But before representing the operators, it is important to illustrate how a population is initiated. As described earlier, a population of chromosomes or individuals is represented by a bit string. The population contains 10 chromosomes. This number is constant through the entire evaluation. Many algorithms operate with pre-programmed chromosomes, this is called "a priori", giving the chromosomes high score values according to prior knowledge. In this thesis all the populations are initiated randomly, they are given the same probability, without any form of knowledge or instinct. The population is then tested towards the problem, in the first generation the score is generally quite low, given the random initiation. The feedback is called fitness measure, or fitness score. After measuring fitness, the selection schemes are to be applied.

4.7.1 Selection

The next step in the evolutionary model is the selection operator. Before going on to the crossover operator and the mutation operator it is time to select which chromosomes are good enough to proceed into the next generation. There are many selection models, in the following section those of current interest are presented.

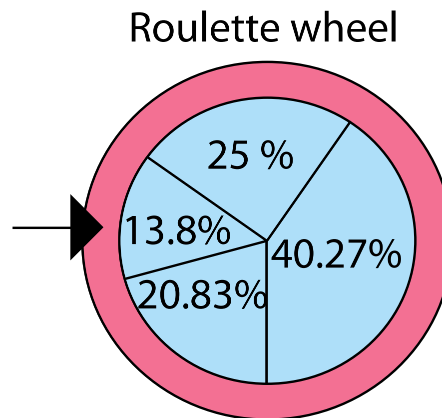

Figure 4.7: *Example of a roulette wheel, 4 chromosomes each have parts of the wheel according to fitness measure.*

Roulette Wheel Selection

All the chromosomes are represented in a roulette wheel [35], illustrated in Figure 4.7. All of the chromosomes are given shares of the wheel according to their fitness score. A chromosome with a score of 3000 will have 3 times as wide a share of the wheel compared to a chromosome with a score of 1000. The wheel spins as many times as there are chromosomes in the new population. The chromosomes with high score, and thus a wide share of the roulette wheel, have a higher probability to be selected into the next generation than a chromosome with a low score and a narrow share of the wheel. Still there is a possibility to be chosen for the chromosomes with low score, this ensures the diversity within the population.

If a population gets stuck in a local optima unable to search widely, diversity is needed. This could happen if a chromosome of extremely high fitness compared to the other chromosomes, is the only chromosome being reselected into the next generation. With diversity, one may find better local optima and maybe the global optima when applying the other operators. If the need for diversity in the population increases there is a possibility to scale the population so that the highest scoring chromosome does not get as big a share of the wheel respectively to the ones with lower fitness score.

Fitness-based Rank Procedure

The ranking procedure [36] is often used for assurance of diversity. After fitness evaluation, the entire population is ranked according to the fitness. The chromosomes are given a new fitness score from 1 to 10 according to its rank. The large differences in fitness value thus disappear. After ranking, a selection model needs to be applied for the selection process (e.g. roulette wheel selection). For further

explanation see pseudo code in the following sections.

Evolutionary Strategie
After using the rank procedure, this method makes sure a certain percentage of the best chromosome are copied up, and selected to proceed to the next generation. A lower percentage of the second best chromosome is allowed to proceed, and so on. In this particular thesis the distribution of percentages that are allowed to proceed is 40 % of the best chromosome, 30% of the next best chromosome, 20% of the third best chromosome and 10% of the fourth best chromosome. For further explanation see pseudo code in the following sections. After selection the crossover and mutation operator are applied to the new population of chromosomes.

Tournament Selection
In this particular selection [36] two random chromosomes are selected to tournament against each other. The chromosome with the highest fitness score wins the tournament and is selected to proceed to the next generation. This procedure is repeated until there are enough chromosomes in the next generation. After selection crossover and mutation are performed. This selection method ensures the diversity in a population.

4.7.2 Elitism/Clone

Elitism (i.e. clone) [37] is important to keep the max fitness score from decreasing. The chromosomes (one or more) with the highest fitness score are directly transferred to the next generation. They are not being transformed with any of the operators. They are kept intact and directly transferred to the next generation.

4.7.3 Crossover

The crossover operator, illustrated in Figure 4.8 takes two chromosomes and splits them into two or more parts. The operator is meant to combine the good qualities of the two parent-chromosomes, making two children-chromosomes, both of them different from the initial ones. Where and how many times the chromosomes are split and combined varies. In this thesis the number of chromosomes to cross also varies. But double-point crossover is used for the chromosomes that are to be crossed, meaning two crossing points in the chromosomes make three parts. The point of crossing within the chromosome strings is random, but every chromosome is to cross with the one below. There is also a phenomenon called knowledge-based crossover, which is optimized for the particular coding depending on the chromosome coding and its meaning. E.g. crossing between the different genes/movements in a chromosome, and not in the middle of a gene/movement, not destroying any of the genes, genes are illustrated in Figure 4.2. Knowledge-based crossover is not used in this thesis. The pseudo code for the crossover operator is shown in the following section.

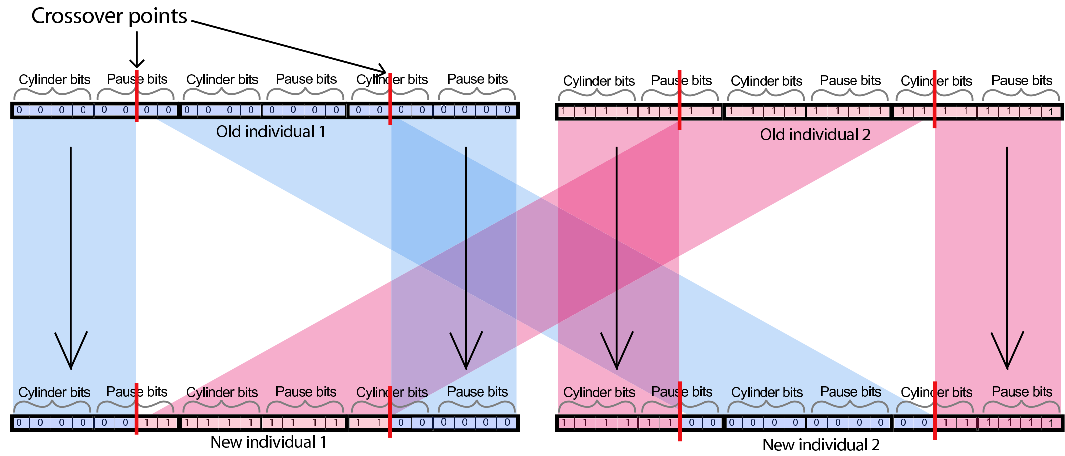

Figure 4.8: *The crossover operator.*

Pseudo code for the Crossover Operator

1. Check whether to perform crossover.
2. If YES: proceed, if NO: stop.
3. Pick a random chromosome.
4. Pick two random crossing point-numbers.
5. Perform crossover on the chosen chromosome and the next chromosome in the places of the random crossing point-numbers.
6. Return the chromosomes to the population.

4.7.4 Mutation

The mutation operator takes a chromosome string, finds one or more of the bits in this string and changes the value. Because we have binary coded chromosome strings containing only '0's and '1's it is easy to just change the selected '0' to a '1', or the selected '1' to a '0' to give the bit the opposite meaning. See Figure 4.9 for illustration. The pseudo code for the mutation operator is shown in the following section.

Pseudo code for the Mutation Operator

1. Check whether to mutate.
2. If YES: proceed if NO: stop.
3. Pick a random chromosome.
4. Pick a random bit within the chosen chromosome.
5. Change the value of the chosen bit within the chosen chromosome to the opposite value.

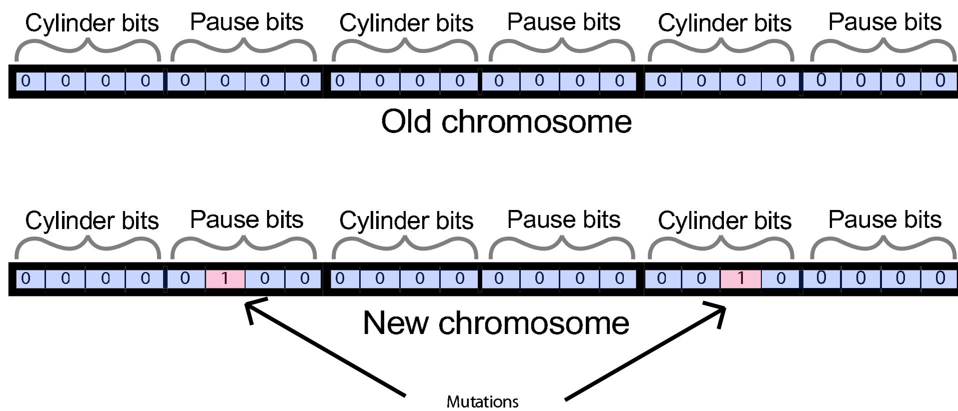

Figure 4.9: *The mutation operator.*

As earlier mentioned all the plots in this thesis consist of 5 graphs, each of them presents an arithmetic mean of the results for each 100 runs. There are 5 of each curve to show the divergence. All chromosomes are also repeated 3 times before fitness evaluation to decrease influence from previous chromosomes and to ensure accuracy of the current chromosome.

4.8 Introducing GA to the MES Robots

Earlier in this thesis it has been demonstrated how well the stochastic search performed in the search space of robotic gaits. The stochastic search found many solutions, and the best ones measured a fitness score of 2015. The next task is to test a simple genetic algorithm on the same problem, to see whether it can manage to find a better solution or not. In this thesis the initial values applied are similar to the default parameters tested by Goldberg [2]. The parameters are shown in Table 4.1. The table shows that the total number of chromosomes is 10, the number of chromosomes to be crossed with another chromosome are 10. This means that all the chromosomes are to be crossed. The mutation rate is also 10. This means that within the population there will be 10 bit mutated, during each generation. There is no use of the elitism, since it does not occur in the Simple GA approach. The plot in Figure 4.10 shows how well the GA performed in the same search space as earlier tested (i.e. finding robotic gaits). The best solution measure a fitness score of over 4000.

CHR	BIT	GENE	GNR	MUT	CROSS	ELITISM	SELECTION
10	8	3	50	10	10	NO	Roulette Wheel

Table 4.1: *Parameters for Figure 4.10 CHR = no. of chromosomes in a population, BIT = number of bits in a gene, GENE = no. of genes in a chr., GNR = no. of generations, MUT = no. of mutations in a generation, cross = no. of chr. to be crossed in a generation, ELITISM = yes/no, SELECTION = type of selection.*

The two plots are shown together in Figure 4.10, with the intention of comparing them to each other. The GA search method is shown in dotted blue lines while the Stochastic search method are shown in dotted red lines. As shown the GA search method rises high above the stochastic in fitness scores. Even the mean fitness score for the GA is somewhat higher than the best fitness score for the stochastic search. But still the GA is too slow to evaluate on real robots. The need for speeding up the GA is high. The focus in the following will be speeding up the GA, and finding good gaits and fast evolution.

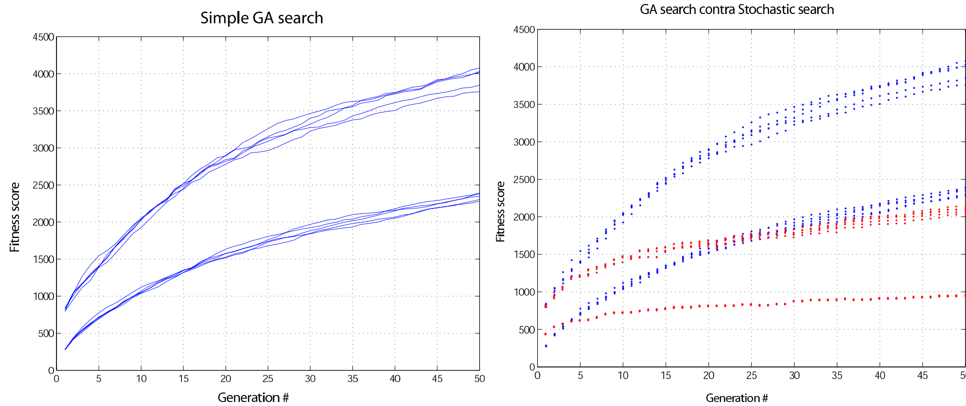

Figure 4.10: *A Simple GA run, shown in blue lines (both solid lines and dotted lines), red dotted lines show stochastic search.*

4.9 Modified GA Runs

According to Goldberg [2] a certain way of speeding up GAs is to add cloning, also called the elitism operator. By simply copying the two chromosomes with the highest score to the next generation without mutating or crossing the genes, it is made sure that the max fitness score will be at least as good as the prior generation. So elitism is therefore added to the GA. The two best chromosomes are directly transferred into the new generation. The parameters are otherwise the same as earlier, as shown in Table 4.1.

MUT	CROSS	ELITISM	SELECTION
10	8	YES	Roulette Wheel

Table 4.2: *Parameters for Figure 4.11, left plot. For number of chromosomes, bits, genes and generations, refer to Table 4.1.*

The results of the GA run with elitism are shown in Figure 4.11, left plot. The simple GA from the last search is still shown in dotted blue lines, while the GA with elitism is shown in solid green lines. Like expected from Goldberg's book, the left plot show an improvement from the simple GA. GA with elitism is of higher fitness score than the simple GA approach. Since the crossover operator tends to destroy already good chromosomes, a run was made with decreased crossover rate. The plot in Figure 4.11, right plot, shows the Simple GA run, still in dotted blue lines, compared to a simple GA run with only 40% crossover rate. The parameters

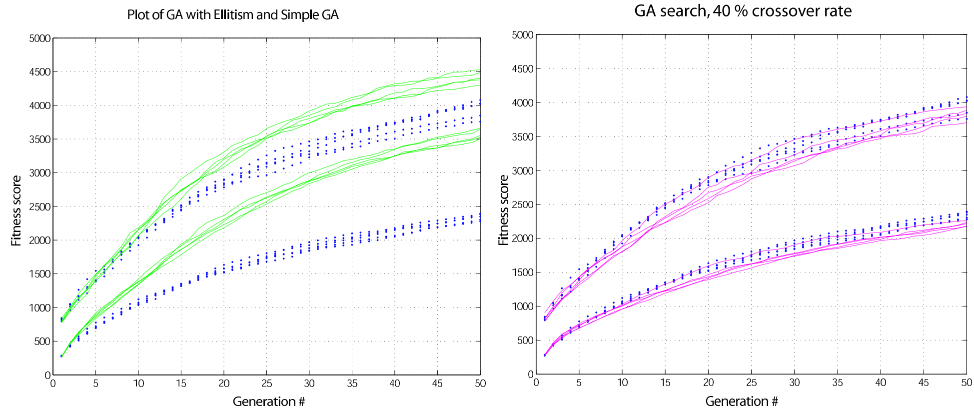

Figure 4.11: *Simple GA, shown in dotted blue lines. Left plot show simple GA contra GA with elitism, show in solid green lines. Right plot show simple GA contra GA where the crossover operator is only performed at 4 of the 10 chromosomes shown in red lines.*

are as shown in Table 4.3. The low rate of crossover did not give a better result in this particular search space. A reason for this is because this run is without the elitism operator, so the relatively low mutation rate was all the difference in the new generations compared to the old generation in addition to the 4 crossed chromosomes.

MUT	CROSS	ELITISM	SELECTION
10	4	NO	Roulette Wheel

Table 4.3: *Parameters for Figure 4.11, right plot. For number of chromosomes, bits, genes and generations, refer to Table 4.1*

The next step was to test a low crossover rate combined with elitism. The plot in Figure 4.12 shows a Simple GA run in dotted blue lines with the same parameters as earlier described, combined with a modified GA run where there are no crossover operator used at all, see Table 4.4, but elitism as described above. This is shown in black solid lines. The results were somewhat better than without elitism and with a low crossover shown in Figure 4.11, right plot, but still not as good as in Figure 4.11, left plot. This is because in this plot the variations are bigger, because of the crossover rate of 4 chromosomes. The idea of increasing the mutation rate therefore arose. Mutation does not destroy a perfectly good chromosome like crossover can do, but it still adds variation to the population.

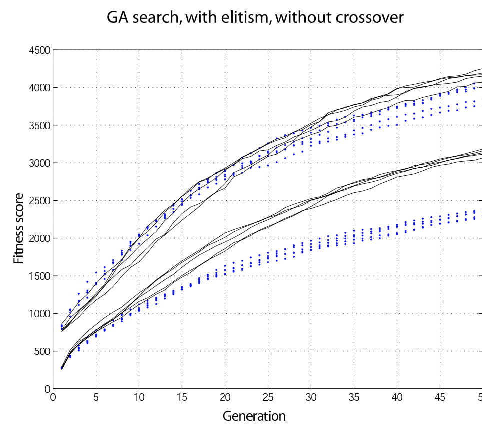

Figure 4.12: *Simple GA (dotted blue lines) contra GA with no crossover, but with elitism as described above(solid black lines).*

MUT	CROSS	ELITISM	SELECTION
10	NO	YES	Roulette Wheel

Table 4.4: *Parameters for Figure 4.12.*

The graphs in Figure 4.13 shows a Simple GA run in dotted blue lines with the parameters in Table 4.1. These graphs are combined with several modified GA runs, where there are no use of the crossover operator and the mutation rate are of different values, see Table 4.5. The GA runs with 300 mutations per population

MUT	CROSS	ELITISM	SELECTION
300/90/50	NO	YES	Roulette Wheel

Table 4.5: *Parameters for Figure 4.13.*

are shown in black crossed lines, the GA runs with 90 mutations per population are shown in solid green lines, the GA run with 50 mutations per population are shown in dashed red lines. Figure 4.13 show large differences in fitness score comparing the different graphs. The best fitness is provided by the plot with 50 mutations per generation. It is better than the simple GA. This is an inexact indication of what the mutation rate should be. It is therefore natural to adjust more accurate. The plot in Figure 4.14 shows the fine adjustments to find the exact best mutation rate for this search space.

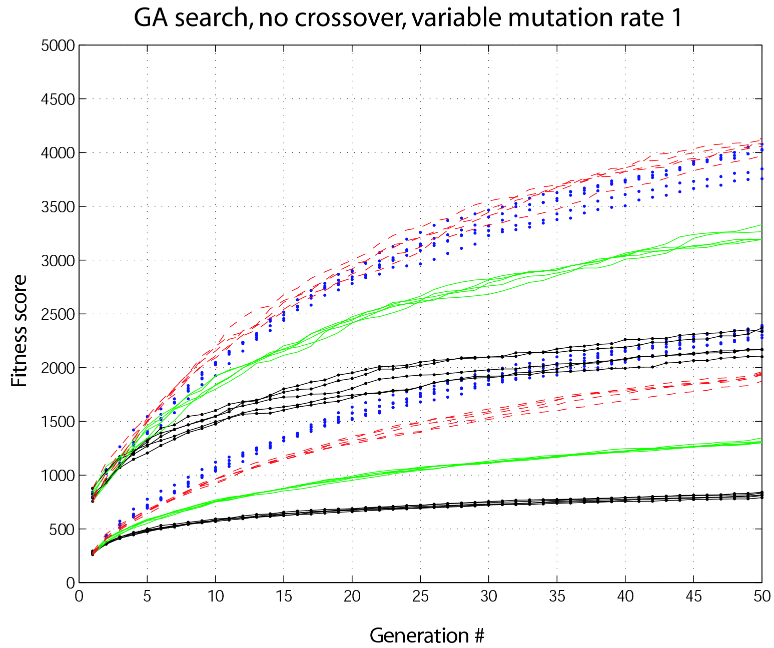

Figure 4.13: *Simple GA is shown in dotted blue lines contra GA-300 mutations in crossed black lines, GA-90 mutations in solid green lines, GA-50 mutations in dashed red lines. Only simple GA uses the crossover operator.*

MUT	CROSS	ELITISM	SELECTION
40/30/**25**/20	NO	YES	Roulette Wheel

Table 4.6: *Optimal parameters (25 mutations) for "roulette-GA" (i.e. GA with roulette wheel selection), Figure 4.14.*

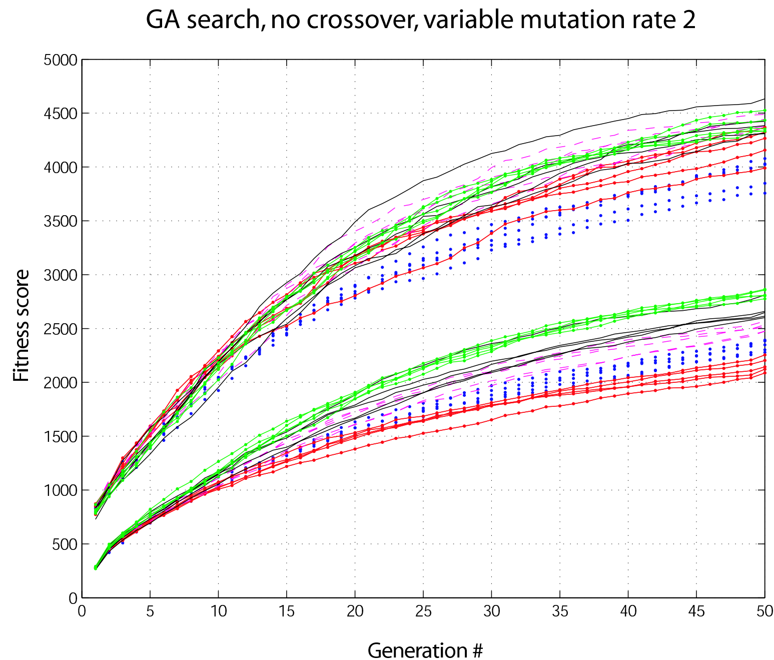

Figure 4.14: *Simple GA is shown in dotted blue lines, contra GA-40 mutations in crossed red lines. GA-30 mutations in pink dashed lines. GA-25 mutations in solid black lines and GA-20 mutations in crossed green lines. Only simple GA uses the crossover operator.*

The plot in Figure 4.14 shows a Simple GA run in dotted blue lines, with the parameters shown in Table 4.1 combined with several modified GA runs where there are no use of the crossover operator and the mutation rate varies, see Table 4.6 All the graphs are of the same fitness score, with insignificant differences. But the graphs showing GA with 25 mutations per population shows a slightly higher fitness score than the other graphs compared in this plot. This is the optimum GA that is found in this thesis by only tuning the crossover and mutation rate, in addition to adding the elitism operator and using roulette wheel selection. The run illustrated in Figure 4.14 in solid black lines, uses no crossing chromosomes, having 25 mutations per population and with elitism, scores the highest. This result will be used as the standard from now on, to compare with other modifications to GA instead of the simple GA, illustrated in Figure 4.10, left plot, that has been used as a standard for comparing so far in this thesis. The new comparable standard will be referred to as the "optimal roulette-GA".

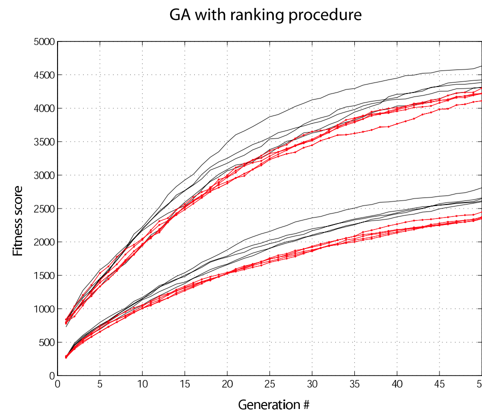

Figure 4.15: *Optimal roulette-GA shown in solid black lines contra GA with ranking procedure and roulette wheel selection shown in red crossed lines.*

4.10 Various Selection Models: Ranking Selection

Pseudo code for the Ranked Selection

1. Initiate a population of chromosomes
2. Test the population towards the search space
3. Get fitness scores for each chromosome
4. Rank the population according to fitness score
5. Give each of the chromosomes new fitness score according to rank (1 - 10), the poorest = 1, the best = 10
6. Perform the Roulette wheel selection until there are enough chromosomes in the next generation
7. Perform crossover / mutation / elitism
8. Go back to number 2

The plot in Figure 4.15 shows optimal roulette-GA in solid black lines. The only selecting operator tested until now have been the roulette wheel selection. The red crossed lines show GA with the same parameters as in black lines, see parameters in Table 4.7. In all the earlier plots the roulette wheel selection is the only selection tested. The next sub chapters will test other selection models for further improvements. This population illustrated in red crossed lines has been ranked before it was reselected with the roulette wheel selection. This ranking procedure may cause

MUT	CROSS	ELITISM	SELECTION
25	NO	YES	Rank & Roulette Wheel

Table 4.7: *Parameters for Figure 4.15.*

a higher probability for diversity in a population. In this plot the diversity was not an advantage to the population, that show a lower fitness score than the optimal roulette-GA.

4.11 Various Selection Models: Tournament Selection

Pseudo code for the Tournament Selection

1. Initiate a population of chromosomes.
2. Test the population towards the search space.
3. Get fitness scores for each chromosome.
4. Pick randomly two chromosomes.
5. The two chromosomes tournament.
6. The one with the highest fitness score wins and is transferred into the next generation.
7. Repeat until there are enough chromosomes in the next generation.
8. Perform crossover / mutation / elitism.
9. Go back to number 2.

4.11.1 Without Elitism

The plots in Figure 4.16, left plot, show optimal roulette-GA in solid black lines plotted contra optimal GA without elitism and with Tournament Selection (TS) operator shown in crossed red lines. As mentioned earlier TS is a selection process where two chromosomes from the current generation are selected to fight each other in a tournament. The chromosome with the highest fitness score wins the tournament, and is selected into the next generation. For further explanation see pseudo code above. The graphs shown in Figure 4.16, left plot have a lower fitness score because there is no use of elitism.

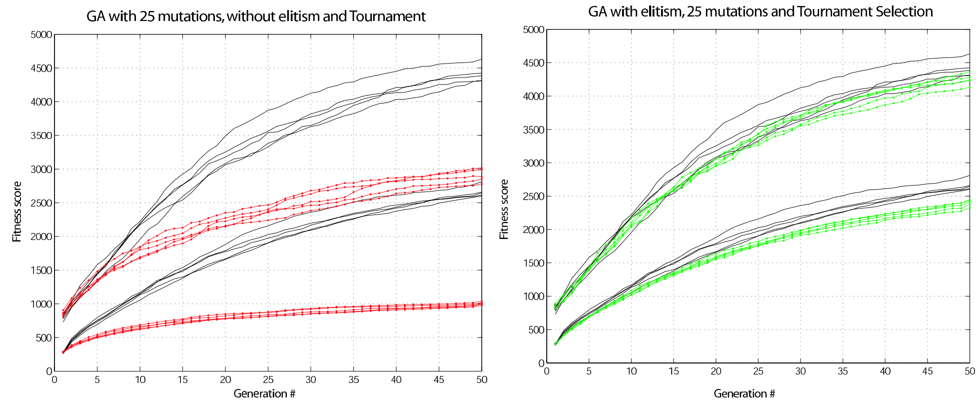

Figure 4.16: *The graphs in solid black lines are the optimal roulette-GA with parameters as shown in Table 4.6. Left plot illustrates GA with parameters as shown in Table 4.8 selected with Tournament Selection (TS) in crossed red lines. Right plot show GA with parameters as shown in 4.9, selected with TS in green crossed lines.*

MUT	CROSS	ELITISM	SELECTION
25	NO	NO	Tournament Selection

Table 4.8: *Parameters for Figure 4.16, left plot.*

4.11.2 With Elitism

MUT	CROSS	ELITISM	SELECTION
25	NO	YES	Tournament Selection

Table 4.9: *Parameters for Figure 4.16, right plot.*

The graphs in Figure 4.16, right plot, show optimal roulette-GA in solid black lines contra green crossed lines that represent GA with elitism and the TS selection method. Although the TS with elitism, green crossed lines, are a lot better than the TS without elitism, black crossed lines in the left plot, they are not better than the optimal roulette-GA shown in black solid lines.

GA with roulette wheel selection may have a problem with lack of diversity in the selection process, but the tournament selection takes care of the problem of diversity by making sure many chromosomes with lower fitness than average are transferred into the next generation. As shown in Figure 4.16, right plot the GA

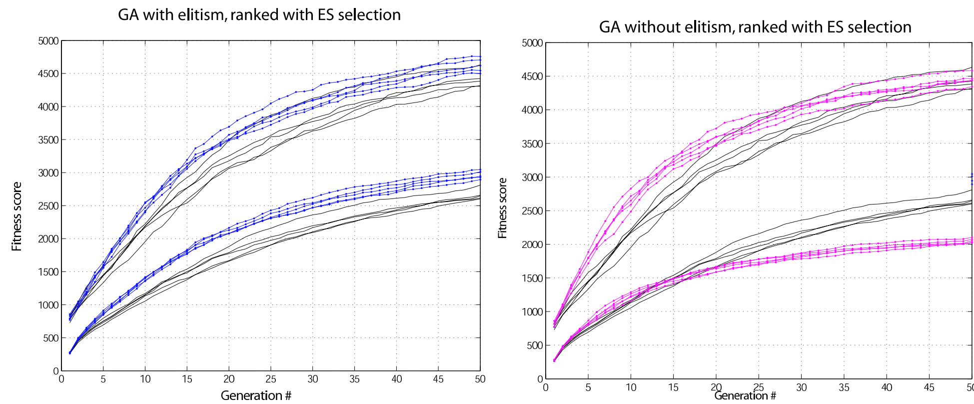

Figure 4.17: *The graphs in solid black lines show optimal roulette-GA. Left plot show GA with elitism, ranked and with ES selection in crossed blue lines, parameters as in Table 4.10 Right plot show GA, ranked and with ES selection but without elitism in pink crossed lines, parameters shown in Table 4.11*

with TS, is less good than the optimal roulette-GA, even if the TS selection with elitism was almost as good as the optimal roulette-GA. This selection model has shown to be less good for this particular problem because of too much diversity. In the next section a new selection model will tested to see if Evolutionary Strategie can match the roulette wheel selection.

4.12 Various Selection Models: Evolutionary Strategie

Pseudo code for the Evolutionary Strategie

1. Initiate a population of chromosomes
2. Test the population towards the search space
3. Get fitness scores for each chromosome
4. Pick the best chromosome, copy it into 40 % of the next generation
5. Pick the next best chromosome, copy it into 30 % of the next generation
6. Pick the third best chromosome, copy it into 20 % of the next generation
7. Pick the fourth best chromosome, copy it into 10 % of the next generation
8. Perform crossover / mutation / elitism
9. Go back to number 2

4.12.1 With Elitism

MUT	CROSS	ELITISM	SELECTION
25	NO	YES	Evolutionary Strategie

Table 4.10: *Parameters for Figure 4.17, left plot.*

In Figure 4.17 left plot, the graphs obtained by optimal roulette-GA are shown in solid black lines. These graphs are illustrated contra GA with the same parameters as the optimal GA, but the graphs illustrated in crossed blue lines are ranked and reselected with Evolutionary Strategie (ES). As shown in the plot this selection method is somewhat better than the Roulette Wheel selection. The ES is coded so that the best chromosome is copied into 40% of the next generation, the next best chromosome is copied into 30% of the next generation, the third best chromosome is selected into 20% of the next generation, and the fourth best chromosome is selected into the remaining 10% of the next generation. Then the mutation operator is performed like described earlier. Elitism is applied on top of the ES. The parameters are as shown in Table 4.10.

4.12.2 Without Elitism

MUT	CROSS	ELITISM	SELECTION
25	NO	NO	Evolutionary Strategie

Table 4.11: *Parameters for Figure 4.17, right plot.*

Figure 4.17, right plot, show the optimal roulette-GA as a comparative, the pink crossed lines represent GA with parameters as in Table 4.11, ranked and reselected with ES, but the elitism is not performed. This plot also show improvement compared to the optimal roulette-GA, but is not as good as the ES with elitism. This is probably because when elitism is applied, the two best chromosomes are directly cloned into the next generation without performing the mutation operator on these two chromosomes. The mutation operator can in addition to create better gaits, also in many ways destroy already good gaits. Therefore by performing the elitism operator, one can save the two best gaits from the previous generation and make sure that the highest fitness obtained does not decrease. The ES is coded the same way as described earlier, for further explanation see pseudo code above.

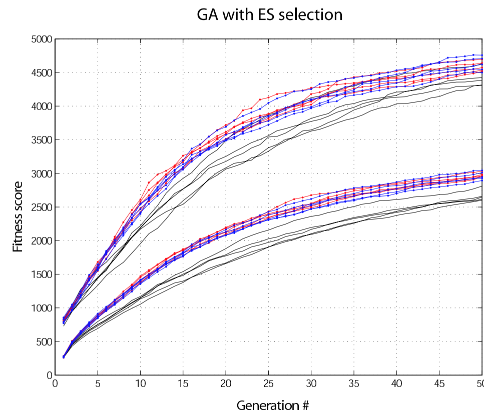

Figure 4.18: *The solid black lines are optimal roulette-GA. The red crossed lines show GA with pauses lasting 0.28 seconds, elitism and the ES selection method is applied, parameters are shown in Table 4.12. The blue crossed lines are GA with ES selection method with variable pauses lasting between 0.05 seconds and up till several seconds, controlled by the GA. Parameters are shown in Table4.12.*

4.13 Testing the ES Selection Model

MUT	CROSS	ELITISM	SELECTION
25	NO	YES	Evolutionary Strategie

Table 4.12: *Parameters for Figure 4.18.*

The plot in Figure 4.18 show optimal roulette-GA in solid black lines. The red lines illustrate GA-ES with fixed pauses lasting 0.28 seconds, while the blue lines illustrate GA-ES with variable pauses, controlled by GA. The differences between the GA-ES with constant pauses and the GA-ES with variable pauses are insignificant. This is probably because the chosen constant pause length is quite optimal for this search space. The plots in 4.18 show that ES selection is better suited for this particular problem than the Roulette wheel selection. Parameters are shown in Table 4.12. The GA-ES with variable pauses shown in crossed blue lines will be referred to as the optimal GA-ES in the next section.

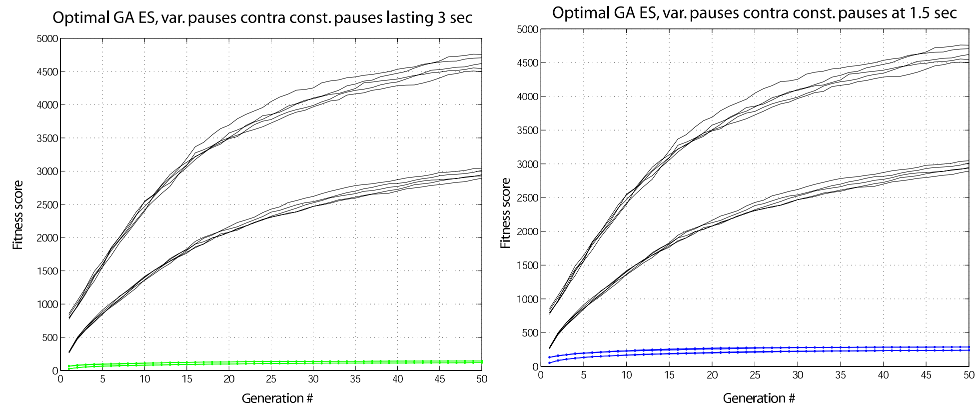

Figure 4.19: *Left plot show optimal GA-ES in solid black lines contra GA-ES with predefined constant pauses set to 3 seconds in green crossed lines. Right plot show the optimal GA-ES plotted contra GA-ES with predefined constant pauses set to 1.5 seconds illustrated in blue crossed lines.*

4.14 Testing the ES Selection Model, Constant Pauses

In this section all the plots contain two plotted graphs. The graphs that are plotted in crossed colored graphs are showing GA-ES with fixed predefined pauses of different lengths. The optimal GA-ES is plotted in solid black lines with variable pauses controlled by GA as a reference. All the graphs have the parameters as described in Table 4.13.

MUT	CROSS	ELITISM	SELECTION
25	NO	YES	Evolutionary Strategie

Table 4.13: *Parameters for all figures with predefined constant pauses in this section.*

Figure 4.19, left plot show the optimal GA-ES with pauses varying within GA in solid black lines contra GA-ES with constant pauses set to 3 seconds in crossed green lines. Right plot show the optimal GA-ES in solid black lines contra GA-ES with constant pauses set to 1.5 seconds in blue crossed lines. Both pauses lasting 3 seconds and the pauses lasting 1.5 seconds obtain a poor fitness score. The performance of the graphs in Figure 4.19, right plot, with half the length of the pauses as in the left plot, have somewhat improved the performance of the GA-ES with constant pauses, but still the results are insufficient for robotic gait evolution and can not compete with the runs done with GA-ES with variable pause lengths.

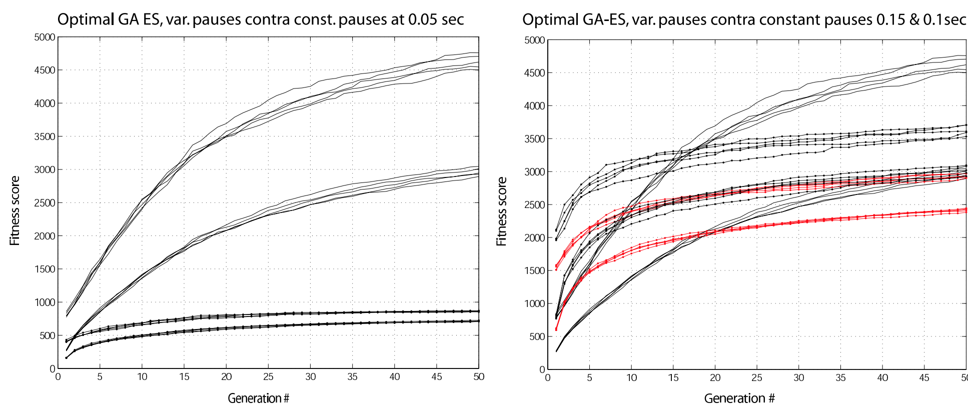

Figure 4.20: *Left plot show optimal GA-ES in solid black lines contra black crossed lines representing the GA-ES with constant pauses set to 0.5 seconds. Right plot show new black crossed lines illustrating GA-ES with constant pauses set to 0.1 seconds and red crossed lines show GA-ES with constant pauses set to 0.15 seconds*

The graphs in Figure 4.20, left plot, show constant pauses lasting one sixth of the length of the pauses in Figure 4.19, left plot. The fitness obtained by the GA-ES with predefined constant pauses is still very poor compared to the optimal GA-ES. The performance in Figure 4.20, right plot, with pauses lasting 0.15 seconds in red crossed lines, and pauses during 0.1, black crossed lines, have increased the fitness score up to an approved level. But still the fitness of the pauses varying within the GA is better in the last 35 generations. In the early generations the GA have improved and show faster evolution of gaits than the optimal GA-ES. This makes it interesting to investigate the meaning of the constant predefined pauses in these first generations since the focus is fast evolution of gaits. The next chapter will contain further developments of this founding.

Based on the research done, it appears obvious that the right pause length causes the evolution to improve, and get a higher fitness score in the first 15 generations simply by setting the right pauses. At the fifteenth generation, however, the plots with variable pauses controlled by GA catches up with the constant pauses. It is obvious that only when the best pause lengths are predefined in advance, the constant pauses are better in the first generations. Otherwise the GA with constant pauses gets a remarkably lower fitness score than the GA with the variable pauses controlled by GA. This is a sort of "a priori knowledge".

Other methods besides tuning the parameters and tuning the pause lengths are needed. This chapter has shown that the lengths of the pauses are an important factor that influences the fitness score. In the next chapter the possibility of further improvements of GA, by incrementally evaluating the pause lengths, are discussed.

Chapter 5

The Incremental Approach

A common problem while programming GA is the size of the search space. When trying to define the problem of desired accuracy, the chromosome that represents the solutions to the search space has a tendency to become too complex and long. This means a lot of time is spent testing in either software or hardware before the GA is properly efficient. As a solution to the time consuming problem, this chapter presents an incremental approach to make GAs more effective. The incremental GA differs from the simple GA because the representation is divided into smaller parts and evolved separately [16] [38]. By gradually evolving a complex task in a series of subtasks, increased complexity can be achieved [39] [40]. This approach can make the evolution faster by sectioning the search space. This means starting with simple behavior and incrementally making the task more complex and general.

5.1 The first Incremental Approach

By optimizing the simple GA and the modified GA with different types of selection models, parameter tuning and pause variation, the gaits still did not evolve fast enough for real time evolution. The fitness development was not efficient enough to operate on an online medium, like a robot. The need for faster methods for gait development emerged, and the incremental idea arose. Several researchers have tested the incremental approach for similar problems with quite successful results. Kalganova [18] proposes a two-step method to decompose a complex problem using incremental evolution. The second step of Kalganovas method is to gradually make the tasks more challenging and general. The method was tested in a digital circuit domain and compared to direct evolution. The results from the experiments show that bidirectional incremental evolution performed significantly better than direct evolution. The evolutionary process changed to the better when different types of decomposition were allowed. Kalganova concluded that this approach could be applicable to many real world applications that consist of a natural hierarchy of behaviors from simple to complex. For further description, see Appendix

Figure 5.1: *By mutating and incrementally maneuvering in the search space, the solutions obtained may vary. Rene Magritte, "Wonders of Nature".*

B. Tørresen introduced an Evolvable Hardware (EHW) architecture for pattern classification with elements of incremental evolution [41]. Experiments with simple GA were applied to find the best possible combination of sub circuits for a prosthetic hand controller circuit. In the first step each motion in the prosthetic hand was evolved separately. In the second step of the incremental approach the motions were evolved simultaneously. A feed-forward neural network was also trained and tested with the same data sets. The results showed that in 59.4% of the tests the average performance of the EHW-architecture did better than the best case of the neural network. Tørresen concluded that evolving incremental systems for complex real world applications could be a promising approach.

There are other researchers who have tested the incremental approach with success. DeJong [38] and Potter [42] have presented cooperative co-evolution. They have successfully applied evolutionary algorithms to the solution of increasingly complex problems while developing effective techniques in the form of co-evolution and co-adaptation. The system architecture allowed the authors to scale up to more complex problems than possible with standard EA. For a more thorough documentation of successful incremental approaches, see Appendix B. From this background it is obvious that incremental approaches have been applied successfully to many real world applications. In the next section the chromosome of the MES robot is incrementally divided, the evolution of cylinder bits are separated from the evolution of pause bits.

By incrementally dividing a search space the number of possible solutions decrease. For the incremental approach to be efficient, it is outmost critical to perform the division in the right manner, otherwise the results can be poor. There is no general form of incremental approach, the incremental solutions must be customized for the current application. Several customized incremental solutions for robotic gaits are presented in the next section.

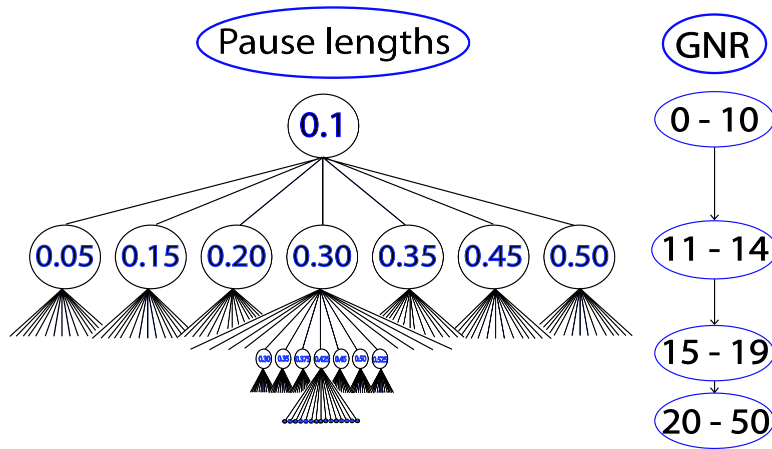

Figure 5.2: *Available pauses in a tree structure, generations shown to the right. As generations evolve, the GA can choose between more predefined pauses. Pauses are given in seconds.*

5.2 Predefined Pauses in a Tree Structure

The incremental approach was first tested by making a tree structure of predefined pause lengths, illustrated in Figure 5.2. The approach is a combination of constant pauses and pauses varying within GA. The idea is to let the pauses vary between two or more predefined pause lengths, letting the GA choose which lengths are the most suitable.

In the first 10 generations all the pauses in a chromosome have the same initial pause length. The initial pause length measures 0.1 seconds. This pause length is the most suitable pause length in early generations according to the tests done with constant pauses in Chapter 4, illustrated in Figure 4.20, right plot. After developing suitable gaits, it is time to see whether the GA can find even better gaits by being able to choose between more pause lengths. When reaching generation number 10, the cylinder bits in the chromosome are set fixed. We now assume that the cylinder bits are quite optimal, but variations in the pause length can still make a great difference between sufficient gaits and good gaits.

At generation number 10, all the pause lengths are initially set to 0.05 seconds. GA now has the opportunity to add additional pause lengths to the original. From generation number 10 the opportunities the GA can choose are 0.05 seconds, 0.1 seconds, 0.15 seconds, 0.3 seconds, or any combination of the listed values. This makes 7 pause lengths, making 7 nodes in the pause tree, varying from 0.05 seconds up to 0.61 seconds.

After reaching the fifteenth generation the GA can also choose to add 0.075 seconds to the number of solutions already presented. This makes $7 * 15$ tree-nodes of pause lengths in the next level.

Again, when reaching generation number 20 a pause length of 0.0375 seconds can be added as a solution to one or more of the prior possibilities, making $7 * 15 * 31$ pause lengths for the GA to choose in the next level. For an illustration of the pause tree see Figure 5.2. For pseudo code how the pause tree is coded see below.

Chromosome coding:

$$\text{chr} = \overbrace{\overbrace{C_1^1 C_2^1 C_3^1 C_4^1}^{cylinder1}\ \overbrace{P_1^1 P_2^1 P_3^1 P_4^1}^{pause1}}^{gene1}\ \overbrace{\overbrace{C_1^2 C_2^2 C_3^2 C_4^2}^{cylinder2}\ \overbrace{P_1^2 P_2^2 P_3^2 P_4^2}^{pause2}}^{gene2}\ \overbrace{\overbrace{C_1^3 C_2^3 C_3^3 C_4^3}^{cylinder3}\ \overbrace{P_1^3 P_2^3 P_3^3 P_4^3}^{pause3}}^{gene3}$$

An explanation of the C_y^x and the P_y^x is as follows, the C stands for bits representing cylinders, while the P stands for bits representing pauses. The x stands for the gene number, the y stands for the bit number. For a more graphical view of the pause and cylinder bits and how the chromosome is represented see Figure 4.4 in Chapter 4.
Pause bit P_1^x represents a pause length of 0.3 seconds. Bit P_2^x represents a pause length of 0.15 seconds. Bit P_3^x represents a pause length of 0.075 seconds and bit P_4^x represents a pause length of 0.0375 seconds. The total pause length for gene x is given by the combination of the corresponding bits. The total pause length will thus be restricted to the $[0.05 - 0.6]$ seconds interval. In the following sections there will be presented two versions of the Pause Tree Structure algorithm.

Pseudo code for Algorithm no.1 with Pause Tree Structure

1. *Generation $\leq$ 10:*
    - Keep all pauses at 0.1 seconds.
    - Run GA on all cylinder bits.
2. *10 $<$ generation $\leq$ 15:*
    - Keep pause bits $P_3^x = P_4^x = 0$.
    - Keep all cylinder bits equal to cylinder bits found in best chromosome, generation 10.
    - Run GA on pause bits P_1^x and P_2^x.
3. *15 $<$ generation $\leq$ 20:*
    - Keep pause bits $P_4^x = 0$.

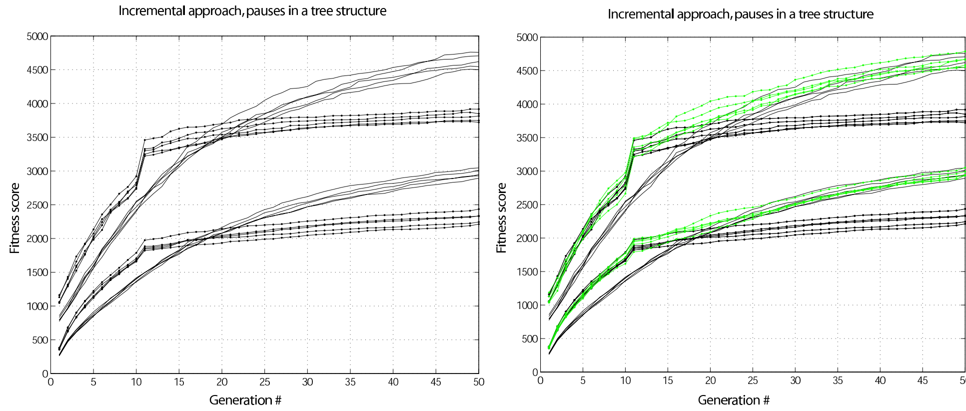

Figure 5.3: *Optimal GA-ES in solid black lines contra Incremental GA, with algorithm no 1 with Pause Tree structure in crossed black lines. The right plot also includes algorithm no 2, where incremental swap between cylinder and pause evaluation is shown in crossed green lines.*

- Keep all cylinder bits equal to cylinder bits found in best chromosome, generation 10.
- Run GA on pause bits P_1^x, P_2^x and P_3^x.

4. *$20 < generation \leq 50$:*

   - Keep all cylinder bits equal to cylinder bits found in best chromosome, generation 10.
   - Run GA on pause bits P_1^x, P_2^x, P_3^x and P_4^x.

5.3 Incremental Results Using Pause Tree.

The GA approach with the pauses in a tree structure is tested towards optimal GA-ES. In the optimal GA-ES the pauses can be of any length, decided by the GA itself, not necessarily be restricted to choose between predefined pause lengths like described in the pause tree structure. The results are shown in Figure 5.3. The optimal GA-ES is shown in solid black lines. The crossed black lines are algorithm no 1, illustrated in Figure 5.2.

Figure 5.3, left plot, illustrates optimal GA-ES in solid black lines. It is plotted contra algorithm no 1, (Incremental GA with the ES selection method) in crossed black lines, both having parameters as showed in Table 5.1. The graphs also illustrate how the incremental approach has been considerable faster in the first 15 generations compared to the optimal GA with pauses varying within the GA. It is obvious that in these first generations the runs that select all possible pause lengths inside

MUT	CROSS	ELITISM	SELECTION
25	NO	YES	Evolutionary Strategie

Table 5.1: *Parameters for Figure 5.3.*

the original optimal GA have a larger search space and therefore the lower fitness in the beginning. But after 20 generations the GA with pause tree structure suffers. This might be because the cylinder bits are fixed without possibility to adjust to the new pause lengths. The next step will be to incrementally swap between cylinder evaluation and pause tree evaluation in algorithm no 2.

Figure 5.3, right plot, illustrates the same graphs as in the left plot, but in addition there are green crossed lines illustrating algorithm no 2, (GA runs with pauses in tree structure.) In the matter of the green crossed graphs the difference from algortihm no1, is that the cylinder evaluation starts again after reaching generation number 15. Algorithm no 2 have the same behavior as algorithm no 1 until generation 15, (i.e. cylinder evaluation up to generation 10). After generation 10 the cylinder bits are kept fixed for the pause tree method to work properly from generation 10 to generation 15. In this way the cylinder evaluation is started again and kept running from generation 15, adjusting to the pauses found for the last 35 generations. The pseudo code for algorithm 2 is shown below.

Pseudo code for Algorithm no 2 with Pause Tree Structure and Cylinder Evaluation

1. *Generation* $\leq$ *10:*
   - Keep all pauses at 0.1 seconds.
   - Run GA on all cylinder bits.
2. *10* $<$ *generation* $\leq$ *15:*
   - Keep pause bits $P_3^x = P_4^x = 0$.
   - Keep all cylinder bits equal to cylinder bits found in best chromosone, generation 10.
   - Run GA on pause bits P_1^x and P_2^x.
3. *15* $<$ *generation* $\leq$ *20:*
   - Keep pause bits $P_4^x = 0$.
   - Run GA on pause bits P_1^x, P_2^x and P_3^x.
   - Run GA on all cylinder bits.

4. *20 < generation ≤ 50:*
   - Run GA on pause bits P_1^x, P_2^x, P_3^x and P_4^x.
   - Run GA on all cylinder bits.

The green crossed lines in Figure 5.3, right plot, are like the black crossed graphs better than the optimal GA-ES in the first 15 generations. In difference from the black crossed graphs, the green graphs stay better than the optimal GA until generation 25 and stays at the same level as, and slightly better than, the optimal GA-ES. This method provides faster evolution of good gaits than the earlier approaches.

5.4 The GA - Binary Hill Climbing Algorithm

The second and more successful incremental approach is the Binary Hill climbing (BH) idea. When the hill climbing approach is applied in a search space, it means the search always proceeds in the direction where the highest fitness is obtained. This search is well suited for search spaces with smooth topography. The GABH algorithm focuses on fast evolution and it lasts only the first 15 generations, as it is targeted at fast online evolution. The Figure 5.4, show the evaluation focus in red color covering the bits being evaluated in each generation.

The first increment of the GABH is very much like the earlier incremental approaches (algorithm no 1 and algorithm no 2). The cylinder bits are evaluated and the pauses are kept constant at 0.1 seconds until generation 10. After generation 10 a hill climbing algorithm is to start the search through the pause search space step by step.

The chromosome with highest fitness from generation 10 is chosen. The cylinder bits in the chromosome ($C_1^x C_2^x C_3^x C_4^x$), are fixed for all further evolution. See illustration in Figure 5.4, generation 11. 8 copies of the best chromosome is made, with all possible combinations of the bits P_1^1, P_1^2, and P_1^3 represented (shown in red color), this is the new population.

These 8 chromosomes are manually tested towards the problem, and fitness is assigned. The best chromosome from generation 11 is chosen to proceed to the next generation. Within this chromosome, the bits P_1^1, P_1^2 and P_1^3 are fixed for all further evolution.

The next step is to release the bits of next most significance (P_2^1, P_2^2 and P_2^3), and keep bits $P_3^x = P_4^x = 0$, see Figure 5.4, generation 12. 8 copies of the best chromosome is made, with all possible combinations of the bits P_2^x. The 8 chromosomes are further manually tested towards the problem, and fitness is assigned. The best

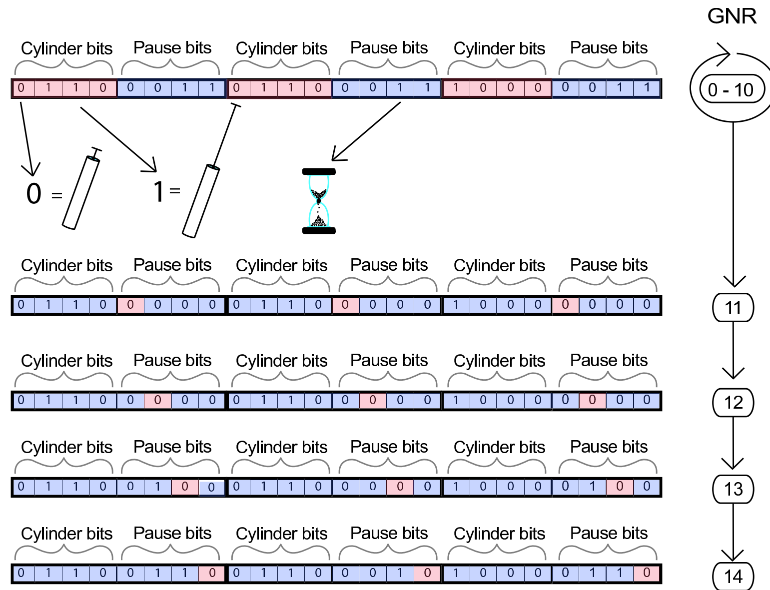

Figure 5.4: *Illustration of GABH, blue color indicates fixed bits, red color indicates evaluation focus in each generation, shown to the right.*

chromosome from generation 12 is chosen to proceed to the thirteenth generation. Within this chromosome, the bits P_2^1, P_2^2 and P_2^3 are fixed for all further evolution.

The process is then repeated for the two last generations. The third most significant bits (P_3^1, P_3^2 and P_3^3) are released, see Figure 5.4, generation 13. The least significant pause bits ($P_4^1 = P_4^2 = P_4^3 = 0$) are kept fixed, see Figure 5.4, generation 13. 8 copies of the best chromosome is made, with all possible combinations of the bits P_3^x. The 8 chromosomes are further manually tested towards the problem, and fitness is assigned. The best chromosome from generation 13 is chosen to proceed to the fourteenth generation. Within this chromosome, the bits P_3^1, P_3^2 and P_3^3 are fixed for all further evolution.

The least significant pause bit P_4^x are released, see Figure 5.4 generation 14. 8 copies of the best chromosome are made, testing all possible combinations of the bits P_4^x. The 8 chromosomes are tested towards the problem, and the best chromosome is preserved. This chromosome will contain the optimal pause length obtained for this exact problem with the GABH algorithm. For further description see pseudo code.

Pseudo code for GABH algorithm

1. *If generation $\leq$ 10:*
   - Set pauses fixed to 0.1 sec.

- Run GA on the cylinder bits.

2. *If generation = 11:*
   - Terminate GA.
   - Select the best chromosome from generation 10.
   - Fix all cylinder bits C_y^x.
   - Set the 3 least significant pause bits to 0 in all 3 pauses.
     $$\text{chr} = \overbrace{C_1^1C_2^1C_3^1C_4^1\ \overbrace{\mathbf{P_1^1}000}^{pause1}}^{gene1}\ \overbrace{C_1^2C_2^2C_3^2C_4^2\ \overbrace{\mathbf{P_1^2}000}^{pause2}}^{gene2}\ \overbrace{C_1^3C_2^3C_3^3C_4^3\ \overbrace{\mathbf{P_1^3}000}^{pause3}}^{gene3}$$
   - Make 8 copies of the chromosome with all $2^3 = 8$ different combinations of the $\mathbf{P_1^1}$, $\mathbf{P_1^2}$, $\mathbf{P_1^3}$ values.
   - Test all 8 chromosomes towards the problem and assign fitness.

3. *If generation = 12:*
   - Select the best chromosome from generation 11.
   - Fix the most significant pause bits P_1^1, P_1^2, P_1^3.
   - Keep the 2 least significant pause bits at 0 in all 3 pauses.
     chr = $C_1^1C_2^1C_3^1C_4^1\ P_1^1\mathbf{P_2^1}00\ \ C_1^2C_2^2C_3^2C_4^2\ P_1^2\mathbf{P_2^2}00\ \ C_1^3C_2^3C_3^3C_4^3\ P_1^3\mathbf{P_2^3}00$
   - Make 8 copies of the chromosome with all $2^3 = 8$ different combinations of the $\mathbf{P_2^1}$, $\mathbf{P_2^2}$, $\mathbf{P_2^3}$ values.
   - Test all 8 chromosomes towards the problem and assign fitness.

4. *If generation = 13:*
   - Select the best chromosome from generation 12.
   - Fix the two most significant pause bits P_1^x and P_2^x in all 3 pauses.
   - Keep the least significant pause bits at 0 in all 3 pauses.
     chr = $C_1^1C_2^1C_3^1C_4^1\ P_1^1P_2^1\mathbf{P_3^1}0\ \ C_1^2C_2^2C_3^2C_4^2\ P_1^2P_2^2\mathbf{P_3^2}0\ \ C_1^3C_2^3C_3^3C_4^3\ P_1^3P_2^3\mathbf{P_3^3}0$
   - Make 8 copies of the chromosome with all $2^3 = 8$ different combinations of the $\mathbf{P_3^1}$, $\mathbf{P_3^2}$, $\mathbf{P_3^3}$ values.
   - Test all 8 chromosomes towards the problem and assign fitness.

5. *If generation = 14:*
   - Select the best chromosome.
   - Fix the 3 most significant pause bits P_1^x, P_2^x and P_3^x in all 3 pauses.
   - Make 8 copies of the chromosome with all $2^3 = 8$ different combinations of the $\mathbf{P_4^1}$, $\mathbf{P_4^2}$, $\mathbf{P_4^3}$ values.
   - Test all 8 chromosomes towards the problem and select the best solution.

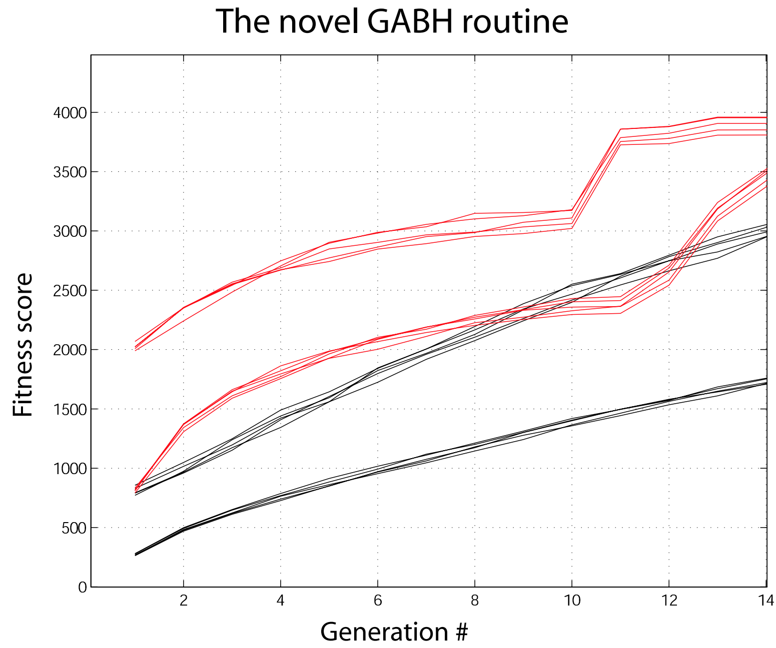

Figure 5.5: *Red lines show GABH algorithm, solid black lines show optimal GA-ES.*

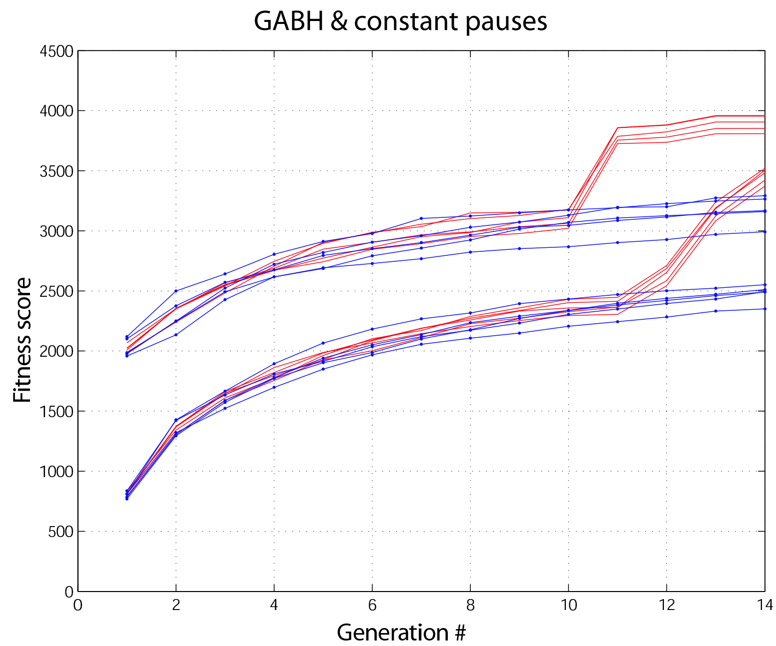

Figure 5.6: *Red lines show GABH algorithm with fixed pauses at 0.1 seconds in the first 10 generations, the crossed blue lines illustrate the optimal GA with fixed pauses at 0.1 seconds.*

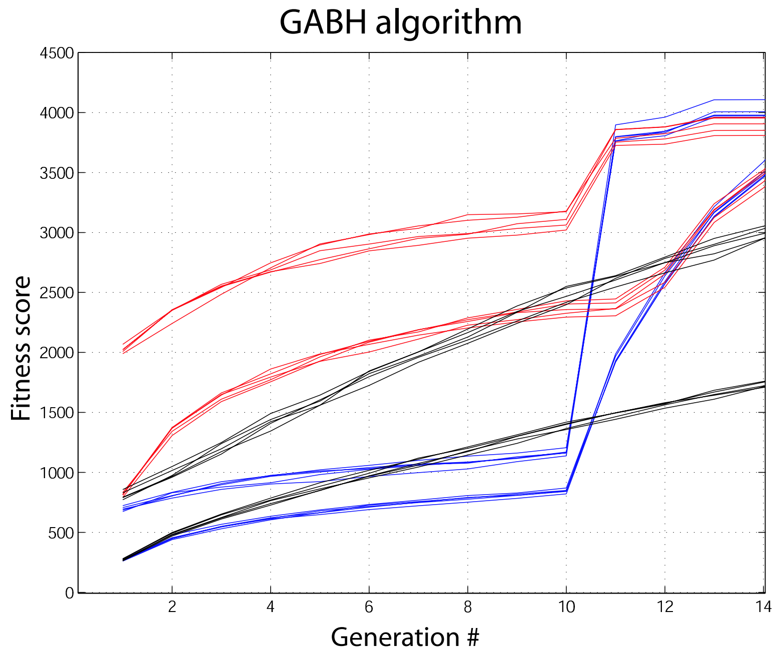

Figure 5.7: *Red lines show GABH algorithm with 0.1 seconds constant pause in the first 10 generations, the solid black lines show optimal GA. The blue lines show the GABH with the constant pauses set to 0.3 seconds in the first 10 generations.*

When the pause search is divided into several parts, the search space for these bits becomes quite narrow. Because the cylinder bits and ¾ of the pause bits are kept fixed in each generation, the opportunity to manually search exhaustively occur in the last part of the GABH algorithm. If the search space was larger it would have been natural to perform random mutations and perform the further search with EAs.

Figure 5.5 illustrates the results of GABH runs. The GABH algorithm is represented in red lines, the solid black lines are optimal GA. In the GABH runs the fixed pauses in the first 10 generations are set to last for 0.1 seconds, which is the best fixed pause in GA runs according to earlier experiments. The GABH runs last only for 14 generations. The GABH algorithm has a higher fitness score than the optimal GA run for 14 generations. This method is more suitable for fast evolution of good gaits.

Figure 5.6 illustrates a plot of the GABH algorithm with fixed pauses at 0.1 seconds, it is compared to the GA with fixed pauses at 0.1 seconds. These graphs are of quite

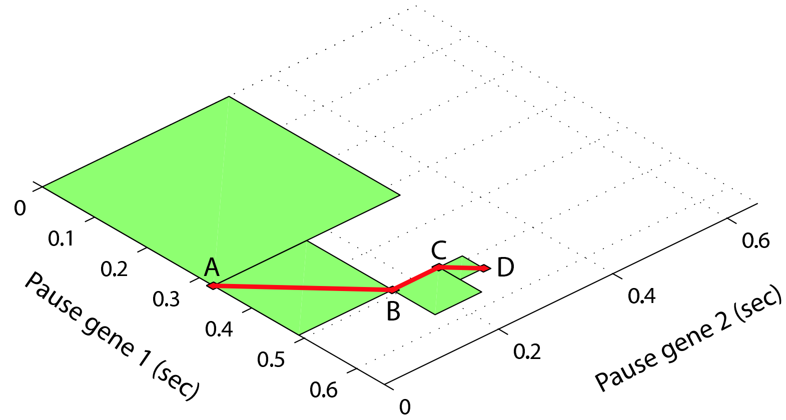

Figure 5.8: *The Binary Hill climbing approach. The 3 dimensional fitness landscape above the squares is not illustrated*

equal fitness score for the first 10 generations, but then the GABH algorithm starts tuning in better suited pause lengths and obtains higher fitness scores.

Figure 5.7 show the GABH run contra the optimal GA-ES (black graphs) and contra GABH with non-optimal initial pause for the first 10 generations (blue graphs). This non-optimal pause is set to 0.3 seconds. Even with this non-optimal pause length the GABH algorithm performs after the tenth generation significantly better than the optimal GA-ES, and even better than the GABH with 0.1 seconds initial pause. The fact that a GABH run with non-optimal initial pause catches up with, and performs better than, the optimal GA-ES and even performs better than GABH algorithm with optimal initial pause, show the stability and reliability of the binary hill climbing algorithm.

To fully understand the operation of the binary hill climbing algorithm one may look at a simplification where the pause in gene 3 is kept constant and the algorithm is applied only to the pauses in gene no.1 and gene no.2. When the pause bits P_1^1 and P_1^2 are varied and the rest of the pause bits are fixed at 0, there are 4 different pause combinations. This is illustrated in Figure 5.8 where the four corners of the largest square represent all four pause combinations. Suppose that the algorithm evaluate the fitness of all 4 corners in the largest square and selects the combination $P_1^1 = 1$ and $P_1^2 = 0$. In the figure this is illustrated by point A.

When $P_1^1 = 1$ and $P_1^2 = 0$ and the pause bits P_2^1 and P_2^2 are varied where the rest of the pause bits are fixed at 0, there are 4 new pause combinations illustrated by the four corners of the next largest square in the figure. Suppose that the algorithm

evaluate the fitness of all these 4 corners and selects the combination $P_1^1 = 1$, $P_1^2 = 0$ and $P_2^1 = 1$, $P_2^2 = 1$. In the figure this is illustrated by point B.

By proceeding with less significant pause bits the algorithm continues to evaluate new squares where each side is half the size of the previous, hence the name "binary hill climbing".

An article about this novel method has been written and is accepted in the GECCO 2006 conference in Seattle this July. The article is to be found in the Appendix A.

Chapter 6

Measured Results

6.1 Gaits Obtained

This chapter addresses methods and results used for both the practical work done with pauses represented by 4 bits in this thesis and the work done with pauses represented by 6 bits in the paper, Appendix A.
The gaits obtained can be parted into three main categories, two suboptimal gaits and one optimal gait. The three types of gaits are presented in Figure 6.1, Figure 6.2 and Figure 6.3. For further illustration of suboptimal gaits see DVD attached in the back of this thesis.

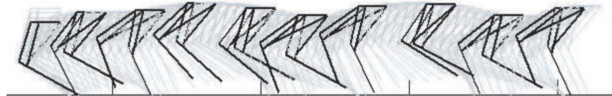

Figure 6.1: *Suboptimal gait based on asymmetric jumping, similar to the fastest horse gait called gallop.*

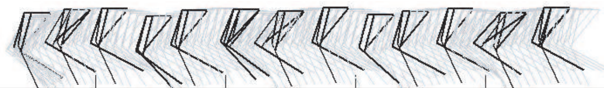

Figure 6.2: *Suboptimal gait based on every other one-leg jumping, similar to the movements made in the sport pole vault.*

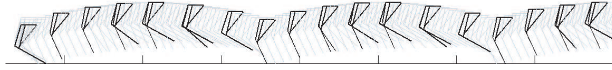

Figure 6.3: *Optimal gaits based on jumping. Most efficient gait obtained, but in real life this gait has many drawbacks (E.g. the slippery effect in the floor to feet friction when the robot kicks hard).*

Figure 6.4: *The pneumatic robot chicken "Henriette" were provided with rubber shoes.*

6.2 Practical Challenges

This section focuses on some of the practical challenges that arose while evolving directly on the robot. The first challenge was the foundation and floor in the laboratory. The floor was too hard for optimal robotic gait evolution and when the robot was expected to jump, it slipped. The robot became worn, due to the hard foundation and vibrations in the balance rod. As a solution, the robot was provided with rubber shoes, illustrated in Figure 6.4. The result was less tear and rod vibrations. Furthermore, the robot began to walk more springier, and started to evolve more efficient gaits based on jumping. The jumping-based gaits turned out to be the most effective. Due to sound propagation, a carpet was needed as a base underneath the robot. The carpet resulted in less noise, but again the slippery effect became an issue. This problem was solved by pasting sand paper underneath the rubber shoes.

Other contributing factors were variations in the air pressure that influenced the performance and the real time qualities. The floor in the laboratory has a slight incline, resulting in a small variety in the fitness measure when evolving on the robot. These descriptions are some of the problems faced when evolving gaits on a real robot.

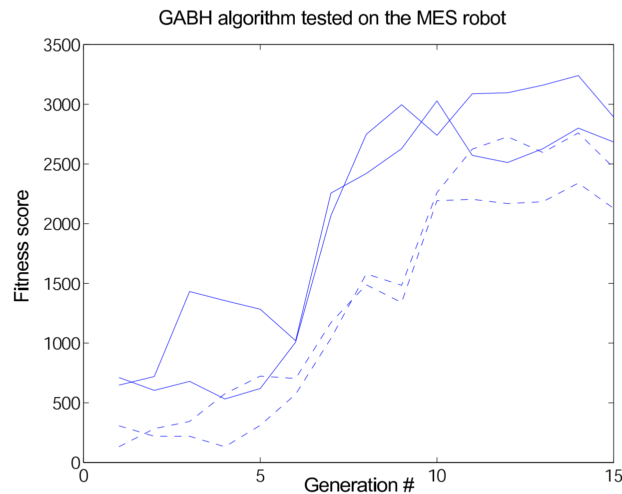

Figure 6.5: *Measured results from GA binary hill climbing approach.*

6.3 Measured Results

The GABH algorithm has been tested on the pneumatic robot in an attempt to verify the theory. Results are illustrated in Figure 6.5. Two typical fitness developments are shown for the GABH algorithm. In these examples the binary hill climbing starting point was set to the seventh generation. From the measurements a fitness improvement is noticeable after this point. After the thirteenth generation the population is kept static, but even for repeated executions of the same chromosomes the fitness found to vary significantly du to practical effects.

It was found very difficult to characterize the physical system accurately due to various practical side effects. One major problem was time consumption and mechanical wear out, particularly of the sandpaper shoe sole which affected the system significantly. When the robot moved, the whole system was vibrating heavily due to the quick contraction/expansion movement of the pneumatic pistons. This vibration made the robot shoe soles occasionally slip during kick-off, and this made the system very unpredictable. Variable friction due to this kind of vibration made the robot occasionally stumble, even for seemingly optimal jumping patterns. Even for repeated executions of the same chromosomes the fitness was found to vary significantly due to practical effects such as variable sole friction. From these few measurements it is difficult to conclude that the GABH algorithm is working significantly better than simple GA. The only conclusion one can make so far from these few measurements is that the algorithm itself is working quite well in this noisy environment. For a further illustration, see the various media contributions in Appendix C and the multi media demonstrations on the DVD.

Chapter 7

Conclusion and proposal for further work

7.1 Conclusion

In this thesis evolutionary algorithms for evolving robotic gaits have been presented. There are several ways to search when trying to find optima, this thesis presents several search methods that have successfully obtained robotic gaits in software and/or in hardware.

The work done in this master thesis is divided into two parts. The first part describes the work concerning the search space given by the 4 bits pause length representation. The other work is presented in the paper to be found in Appendix A, representing the pauses by 6 bits. The search spaces and which method found optimal differs slightly from the one to the other. Both simulations and measurements have been carried out. In simulations the results of the different approaches have been easy to differentiate from each other. In measurements GA has shown to be effective at finding good gaits, but due to the practical challenges it has been difficult to measure the difference between each algorithm. The methods tested for obtaining robotic gaits are listed in the following section.

- **Stochastic search** was tested as a comparative reference. It was tested on both search spaces.
- **Simple GA** approach was tested, performing considerably better than the stochastic approach. This approach was tested on both search spaces.
- **Modified GA** was found optimal when there was no crossover, 25 mutations per generation, extensive use of elitism and evolutionary strategie was used as the selection method in the work where the pauses was represented by 4 bits.

In the matter of the 6 bit represented pauses, the work done with the parameter tuning was found optimal at no crossover, 48 mutations per generation and with roulette wheel selection.

- **GA fixed pauses** was tested only in the search space with the 4 bit pauses, when lasting 0.1 seconds the results was found optimal.
- **Incremental GA** was tested on both search spaces. By separating the cylinder bits from the pause bits the fitness score increased, for the work done with 4 bit pause representation. The incremental approach obtained the better results than all the GA approaches for the 4 bits pause representation. For the 6 bits pause representation the incremental approach never obtained better results than the fitness provided by simple GA.
- **GABH** was tested on both search spaces. For the work done with 4 bits pause representation, the GABH provided the best score after the 14 generation, compared to the other methods at the 14 generation. But all GA approaches scored higher after 50 generations, than the GABH after 14 generations. For the work done with 6 bits pause length representation however, the scores obtained after generation 15 was in average superior to the other applications, even the fitness obtained by the other methods at generation 50.

A bar chart showing all the different search methods tested for the 4 bits pause representation is illustrated in Figure 7.1 to compare the different methods to each other. This overview show the fitness measures after generation 14 shown in light blue color, and the fitness measure after generation 50 shown in purple color. The different search methods are illustrated in ascending order, sorted by the fitness score obtained after the fourteenth generation. Stochastic search followed by the simple GA approach are shown farthest to the left, while the method farthest to the right is the GABH algorithm, who have the highest obtained fitness score after 14 generations. The GABH algorithm was not tested for more than 14 generations.

A bar chart showing the different search methods tested for the 6 bits pause representation is illustrated in Figure 7.2 to compare the different methods to each other.

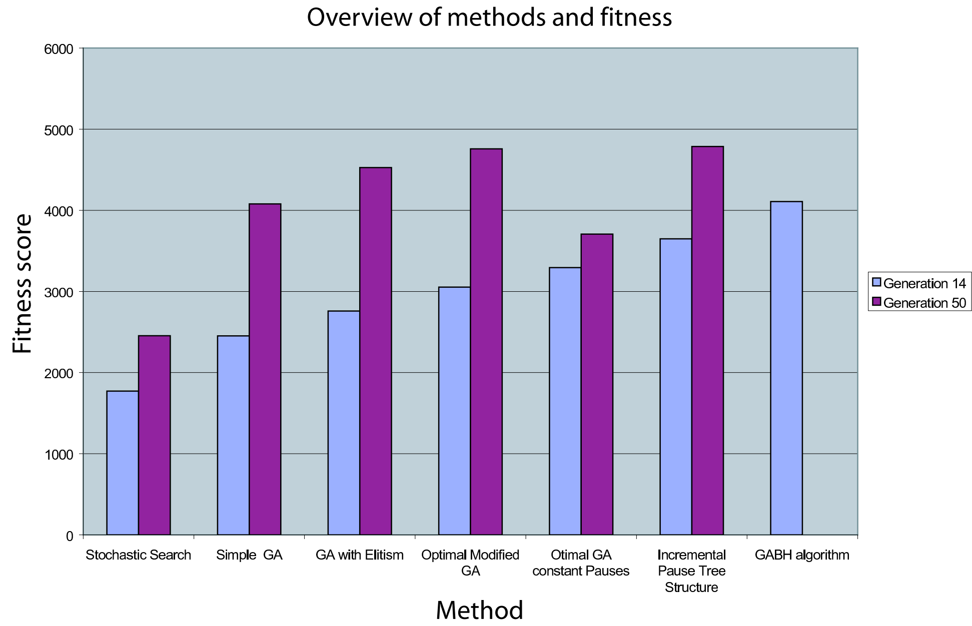

Figure 7.1: *Summary of the search methods tested in the search space with 4 bits pause representation. The methods are sorted ascending by the fitness obtained in generation 14.*

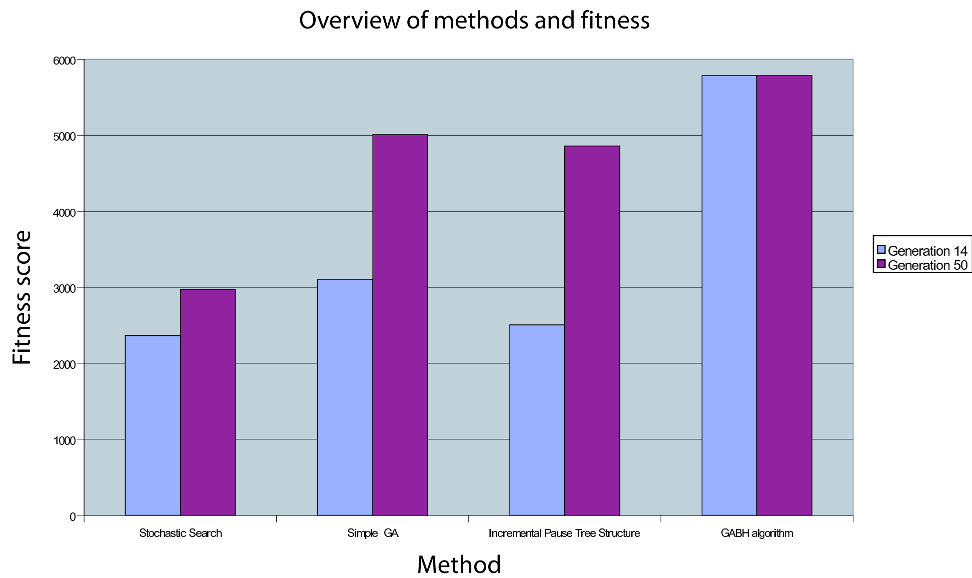

Figure 7.2: *Summary of the search methods tested in the search space for the 6 bits pause representation.*

7.2 Discussion

Several search methods have been tested in a chaotic search space for finding efficient robotic gaits. The focus besides getting efficient gaits has been fast learning.

Stochastic search, simple GA with elitism and modified GA was successfully tested. Due to practical problems regarding both hardware and the environment variables, a simulator approach was enforced, saving a lot of time and preventing the further robotic wear out.

An incremental search algorithm combining GA and binary hill climbing has been presented. In various simulations this algorithm has shown to develop accurate walking patterns faster than standard GA/ES based algorithms. However, in a physical environment with practical side effects such as unpredictable shoe sole friction, varying pneumatic air pressure and wear out, it has been difficult to prove that this algorithm is better than standard GA based algorithms when evolving in hardware. The algorithm itself is, on the other hand found to perform quite well in a noisy environment.

7.3 Other Ideas

This section contains ideas that have been invented, discussed and some of them are tested through out the time spent working with this thesis. The tested ideas have however not been as efficient as hoped and expected, or other problems have occurred and interfered with the efficiency of these ideas. These ideas have therefore not been published like the GABH algorithm.

7.4 Idea: Mother Routine

A lot of time was spent tuning parameters to find the optimal GA, the idea of making a "Mother Routine" came up, this method would run in the background and provide GA with new sets of parameters. The routine would feed GA with parameters according to efficiency and fitness measures. For the "Mother Routine to work properly, it would actually be necessary to run some kind of GA within the "Mother Routine" as well. This routine would result in the robot being able to walk and test different parameters for a very long time without human interference. But unfortunately the wear on the robot restricted the possibilities to accomplish this task. It is probably better to apply this routine for gait simulations than on a real robot performing online. This routine is more thoroughly described in the further work.

7.5 Idea: Positive Reinforcement

The feedback given from the robot is a very accurate measure. Varying this feedback can make the robot seem more alive or responsive to a live audience. A possibility is to let an audience give the feedback, (the robots have been performing for live audiences quite a lot). Several sensors could be attached, for instance sensors

Figure 7.3: *Illustrating positive reinforcement*

sensitive to sound. I.e. if the robot performed in ways the audience percepted as good, and the audience responded with hand-clapping and made a lot of noise, this could give a high fitness score and this particular motion could be repeated. This is a very inaccurate measure, and there would also need to be a delay from the robot have behaved optimal until the audience responds to the behavior and the feedback is registrated by the robot. This method could be a interesting way to make a robot respond to an audience, by repeating and evolving those individuals what the audience think is good behavior.

7.6 Idea: Control Method / A priori Knowledge

The idea behind this method is to keep chromosomes or series of good chromosomes in a library. This means making an "a priori" library of good patterns earlier evolved. One of the drawbacks in Genetic Algorithms and related programming methods are the possibility to end up in local optima without finding optima with higher fitness. A possible way to expand this thesis would have been to make some sort of library of chromosomes that have been found favorable. If the GA gets stuck in local optima with low fitness, new chromosomes from the library could have replaced some of the chromosomes in the population. This routine can be a "Control Method" that runs in the background replacing individuals if the mean fitness does not exceed a certain level. This control method could even be combined with the "Mother Routine".

7.7 Idea: Cyclic

Although the search space can be made slightly smaller by representing each gait by a cyclic coding [43] our experiments have shown no noticeable difference in search speed for cyclic/non cyclic coding for this robot. The size of this search space clearly requires a more efficient search algorithm than simple GA in order to enable cyclic real-time gait development in hardware.

7.8 Idea: Reinforced Learning

Q-learning (RL) have also been tested unsuccessfully in hardware. The results obtained were poorer than any run with GA. The testing was only performed in hardware.

7.9 Further work

There are several things that would have been done differently if this thesis was to be done all over again. The main thing would have been to code the GA in **C++ instead of C**. The Object orientation would have advantages when the program source code gets large and complex and difficult to follow.

There are great opportunities to further develop the GABH algorithm, let it last for more generations by representing the pauses with more bits. This would of course make the search space larger as well.

Regarding simulations the **"Mother Routine"** could have found even better combinations of parameters in simple GA and modified GA. The "Mother Routine" combined with a **"Control Method"**, a library of "A Priori Knowledge", could probably improve the GA a lot. But then it would have been a different type of GA when applying the "a priori knowledge".

If the robot is to perform for live audiences, or in other robots that may have other types of feedback it could have been visually funny to apply **Positive Reinforcement** in forms of sensors to make the audience give the feedback. The limitations here are several, as there would have to be a delay from the robot performs till the program receives the feedback from the audience. There would also have to be a noise filter, because the audience would probably make noise no matter if the robot behaved good or the poorly. There is no guarantee that the audience would behave in ways that are suited as GA feedback, but if this worked it would have been a success for the audience to see the robot respond to their response.

Appendix A

GECCO 2006, Published Paper

GECCO: Genetic and Evolutionary Computation Conference

Robot Gaits Evolved by Combining Genetic Algorithms and Binary Hill Climbing

Lena Mariann Garder
Department of Informatics
University of Oslo
N-0316 Oslo, Norway
lenaga@ifi.uio.no

Mats Erling Høvin
Department of Informatics
University of Oslo, Norway
N-0316 Oslo, Norway
matsh@ifi.uio.no

ABSTRACT

In this paper an evolutionary algorithm is used for evolving gaits in a walking biped robot controller. The focus is fast learning in a real-time environment. An incremental approach combining a genetic algorithm (GA) with hill climbing is proposed. This combination interacts in an efficient way to generate precise walking patterns in less than 15 generations. Our proposal is compared to various versions of GA and stochastic search, and finally tested on a pneumatic biped walking robot.

Categories and Subject Descriptors

I.2.9 [**Artificial Intelligence**]: Robotics—*Propelling mechanisms*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms

Keywords

Evolutionary robotics, Genetic algorithms, Machine learning

1. INTRODUCTION

Evolutionary algorithms has often been proposed as a method for designing systems for real-world applications [6]. Developing effective gaits for bipedal robots is a difficult task that requires optimization of many parameters in a highly irregular, multidimensional space. In recent years biologically inspired computation methods, and particularly genetic algorithms (GA), have been employed by several authors. For instance, Hornby et al. used GA to generate robust gaits on the Aibo quadruped robot [7]. GA applied to bipedal locomotion was also proposed by Arakawa and Fukuda [1] who made a GA based on energy optimization in order to generate a natural, human-like bipedal gait. One of the main objections to applying GA's in the seach for gaits is the time-consuming characteristic of these techniques due to the large fitness search space that is normally present. For this reason most approaches have been based on offline and simulator based searches. To reduce the time spent searching large search spaces with GA, various techniques for speeding up the algorithm have been presented.

With the increased complexity evolution schema introduced by Torresen [11], Torresen has shown how to increase the search speed by using a divide and conquer approach, by dividing the problem into subtasks in a character recognition system. Haddow and Tufte have also done experiments with reducing the genotype representation [5]. Kalganova [9] has shown how to increase the search speed by evolving incrementally and bidirectional to achieve an overall complex behaviour both for the complex system to the sub-system and from sub-system to the complex system. For an exhaustive description of other approaches readers may refer to Cantu-Paz [2].

The robot presented in this paper is a two-legged biped with binary operated pneumatic cylinders. The search space in our experiments was set up to describe the forward speed of the robot given the different gaits, and the goal was to find the most efficient gait with respect to speed. To enable efficient gaits the search space needed to be quite large as the accuracy of the pause lengths between the different leg positions is outmost critical, especially for gaits dominated by jumping movements. The focus has not been on evolving a balancing system as there have been no other sensory feedback than the forward position of the robot.

The main goal for our work was to find a search algorithm fast enough to enable real-time gait generation/adaptation where the fitness is provided by the mechanical robot without the need for an offline simulator model.

In this paper we present a different approach to increase the search speed by combining GA and binary hill climbing (BH) in an algorithm that we will refer to as the GABH algorithm.

In chapter II we describe the robot hardware and in chapter III we describe how the different gaits are represented in the chromosome. In chapter IV we present the simulated results of different search algorithms compared to the new GABH algorithm, and in Chapter V we present measured results of the GABH algorithm applied to the hardware robot in real-time with no simulator model.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'06, July 8–12, 2004, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-186-4/06/0007 ...$5.00.

2. THE ROBOT HARDWARE

The robot skeleton is made of aluminium and is provided with two identical legs. The height is 40 cm. Each leg is composed of an upper part (i.e. the thigh) connected through a cylindrical joint to the lower part (i.e. the calf). Pneumatic cylinders are attached to the thigh and the calf used for controlling the movements of the calf and the thigh separately. As shown in Fig. 1 the rear cylinder in each foot actuates the calf whereas the front cylinder actuates the thigh. The cylinders can either be fully compressed or fully extended (binary operation), and the pneumatic valves are located on top of the robot. The valves are electrically controlled by 4 power switches connected to a PC I/O card (National Instruments DAQ-pad) and the different searching algorithms are implemented in the programming language C++ on the PC.

The pneumatic air pressure was set to 8 bar and provided by a stationary compressor. The robot was attached to a balancing rod at the top (Fig. 1 right and Fig. 2) making the robot able to move in two dimensions. The other end of the rod was attached to a rotating clamp on a hub. The robot walks around the hub with a radius of 2 meter. In addition to being a balancing aid, the rod supplies the robot with air pressure and control signals from the DAQ-pad. The hub has a built in optical sensor representing the rod angle in 13 bit Gray code.

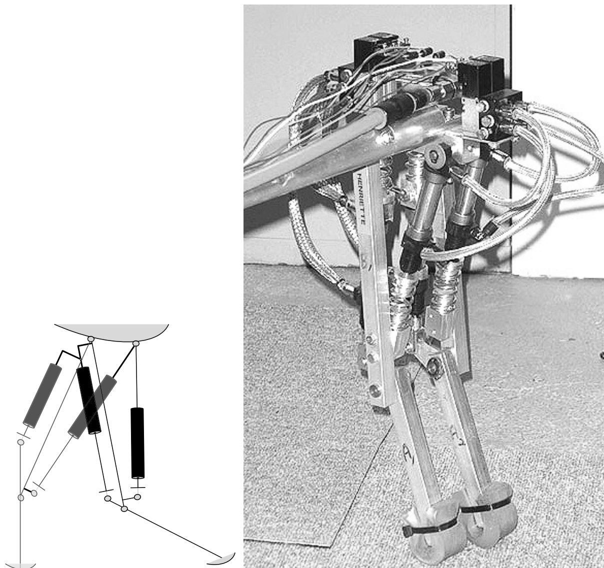

Figure 1: Illustration (left) and photo (right) of the robot. Proper walking direction is left to right (bird construction).

3. GENETIC ALGORITHM

3.1 Simple GA

A genetic algorithm is based on representing a solution to the problem as a genome (or chromosome). The genetic algorithm then creates a population of solutions and applies genetic operators to evolve the solutions in order to find the best one(s). In the simple GA approach [4], [12] the chromosomes are randomly initiated and the only genetic operators used are mutation and crossover. The selection process is done by roulette wheel selection.

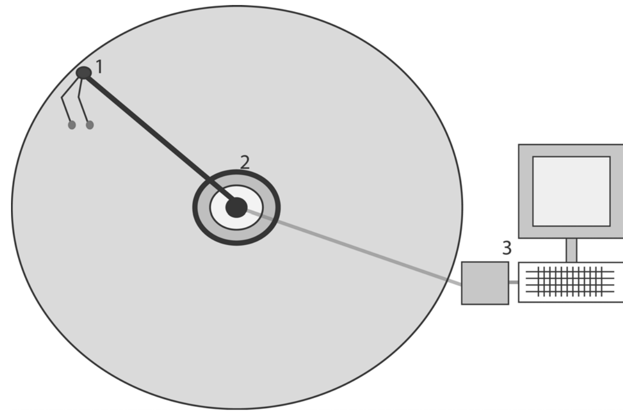

Figure 2: The fitness measurement and balancing rod system (top view).

3.2 The chromosome coding

In our experiments each gait is coded by a 30 bit chromosome. The chromosome represents three body positions each followed by a variable pause. A body position is composed of the positions of the 2 legs (4 cylinders) and represented by four bits (Fig. 3) each describing the status of the corresponding cylinder (compressed or extracted). A complete gait is then created by executing 3 body positions with 3 appropriate pauses in between. Each pause length is represented by 6 bits. The pause length is represented as a binary number corresponding to pauses from 50ms to 300ms. Various simulations have shown no GA search speed improvement by representing the pauses in Gray code.

Two cylinders can move a single leg to 4 different positions. Two legs with four cylinders can hold 16 different positions, and three following positions with 6 bits pauses in between make a search space of

$$2^{30} = 1073741824 \tag{1}$$

different gaits.

Although the search space can be made slightly smaller by representing each gait by a cyclic coding [10] our experiments have shown no noticeable difference in search speed for cyclic/non cyclic coding for this robot. The size of this search space clearly requires a more efficient search algorithm than simple GA in order to enable real-time gait development in hardware.

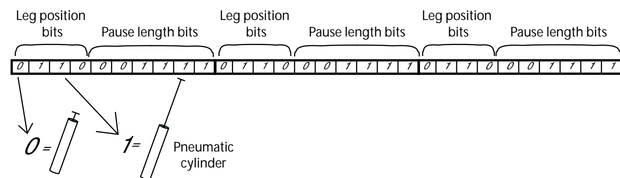

Figure 3: The chromosome internal coding.

3.3 Pauses

A gait is composed of leg positions and pauses. In our robot evolution we have found that the most efficient gaits with respect to forward speeds are gaits dominated by jumping movements. In a jumping movement the pause length between each leg kick is outmost critical as the robot may stumble if the timing of the leg kick is just slightly wrong. Measurements show that a pause length deviation in the magnitude of 10ms can make the difference between a relatively useless and a highly effective gait. It is however a trade-of between the desire to represent the pause lengths with a high number of bits and the exponential decrease in search speed for each extra bit used due to the increased size of the search space.

4. SIMULATED RESULTS

To compare the efficiency of the different search algorithms against each other the robot was first simulated in software.

4.1 The simulator

A simple mechanical chicken-robot simulator has been implemented in C++. This simulator models the robot with exact physical dimensions and a weight of 3kg. The centre of gravity is located at the hip joint. It was found very difficult to model the feet-to-floor friction force exactly as this force is heavily modulated by large vibrations in the robot body and supporting rod during walking/jumping. The feet-to-floor friction force is a very important factor for developing efficient jumping patterns and the lack of an exact model for this effect is assumed to be the main weakness of the simulator. The fitness of each chromosome (gait) is a function of the forward speed of the robot caused by the corresponding chromosome. Each gait is repeated 3 times in sequence to reduce the impact caused by the initial leg positions. A movement in the backward direction causes the fitness to be zero.

4.2 Search space topology

The optimal search algorithm for a given problem depends heavily on the topology of the search space. For the chromosome coding described in chapter 3.2 and the chosen software robot model we have tried to get an overview of this topology by separating the search space in two parts, one part generated by the pause bits and one part generated by the leg position bits.

Fig. 4 shows a plot of the fitness landscape for all possible leg positions in a single chromosome (gait) were all 3 pause lengths are fixed at 100ms. The size of this search space is $2^{4 \cdot 3} = 4096$ leg positions. This plot indicates that the part of the overall search space generated by the leg positions is very chaotic although there may be some repetitive phenomena. A similar topology has been found for other choices of constant pause lengths. The different leg positions are sorted by the Gray value of their corresponding bits to keep the bit difference between neighbouring chromosomes in the plot as low as possible, but even so the landscape is chaotic with many narrow peaks.

In Fig. 5 the fitness landscape is plotted for different pause lengths where the leg positions are kept constant. To make the fitness landscape visually informative one of the 3 pause lengths are also kept constant at 70ms resulting in a three dimensional plot. As this plot indicates the part of the overall fitness landscape generated by the pause lengths is smooth and will typically contain a few numbers of maxima. In this type of landscape a hill climbing search will normally be more efficient than a genetic algorithm.

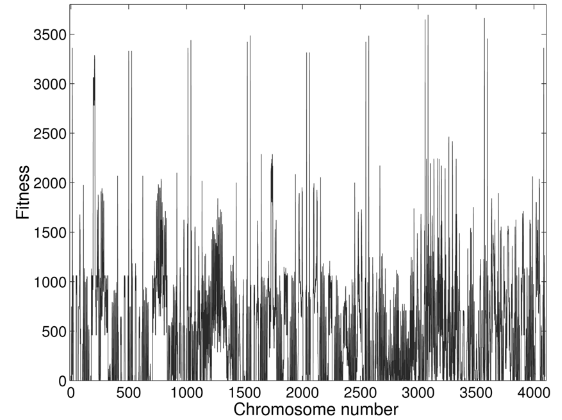

Figure 4: Fitness search space for different leg positions (fixed pauses at 100ms).

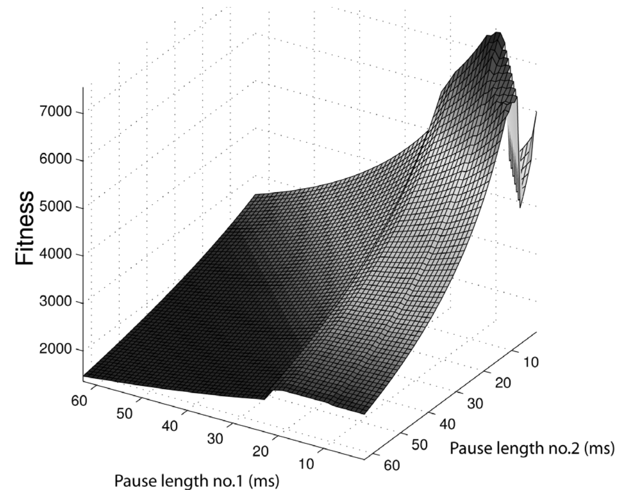

Figure 5: Fitness search space for pause no. 1 and no. 2. All leg positions and pause no. 3 are fixed.

4.3 Simple GA simulations

The focus for this real-time application has been to find a search algorithm capable of finding an optimal gait in less than 20 generations. The first search approach was to perform a search for an optimal chromosome (gait) in the global search space consisting of 2^{30} different chromosome values. Simple and more advanced genetic algorithms were tested against different evolutionary strategies (ES) [4]. ES's showed to be less effective for this particular application and a genetic algorithm was therefore chosen.

In all our simulations 5% noise is added to the fitness function to model practical effect such as variable foot friction,

vibrations, variable air pressure and pause length deviations caused by non-ideal real-time behaviour of the XP operating system.

A simple genetic algorithm with roulette wheel selection, elitism, a population size of 10 chromosomes, no crossover but with as high as 0.2% mutation probability for each bit was found to be the most effective. The high mutation probability indicates that GA is struggling with the topology in this global search space. This result is not surprising as the global search space is assumed to be dominated by the chaotic and complex phenomena shown in the partial search space shown in Fig. 4. In Fig. 7 we see that GA produces slightly less than twice as effective gaits compared to a stochastic search after 15 generations. In all plots each graph shows the mean result from 1000 simulations with randomly initiated populations. 5 different graphs are shown to illustrate the consistency of the simulations.

4.3.1 An incremental GA approach

The next approach was to evolve the partial search spaces shown in Fig. 4 and Fig. 5 separately by an incremental genetic algorithm. Incremental GA differs from simple GA because the search space is divided into smaller parts and evolved separately [11] [8]. By gradually evolving each task in series increased complexity can be achieved [3] [1]. The first incremental approach was to first evaluate the leg position bits, with fixed pause lengths. After obtaining gaits with sufficient fitness the leg position bits are fixed and the pause bits are evolved separately. From Fig. 6 we se that this approach is not successful as the fitness is never found to be higher than the fitness provided by simple GA. Leg position bits are evolved up to generation 11 and pause bits are evolved from generation 12.

The next incremental approach was to divide the search in to 7 increments. First the leg position bits were evolved, then the most significant pause bits were evolved, then the next most significant pause bits were evolved until the least significant pause bits were evolved in the last increment. Even this approach was not found to provide better results than simple GA.

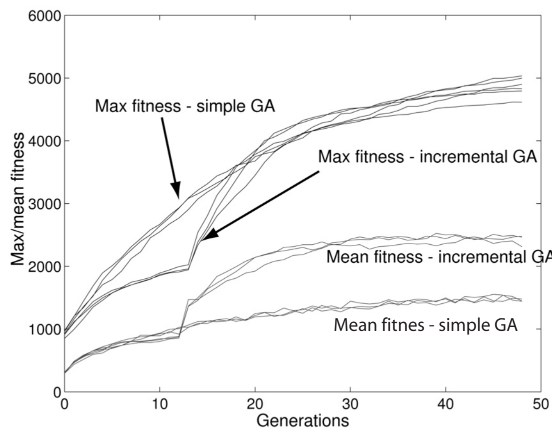

Figure 6: Incremental GA versus simple GA. Leg position bits are evolved up to generation 11 and pause bits are evolved from generation 12.

4.4 The GABH algorithm

The third and more successful incremental approach was to combine GA and binary hill climbing in the GABH algorithm. From Fig. 5 we notice that the typical pause length fitness landscape is smooth with few maxima. In a practical application disturbances will be added to this landscape due to variable foot friction, vibrations, variable air pressure and pause length deviations caused by non-ideal real-time behaviour of the operating system. However, the main characteristic of this landscape indicates that a hill climbing algorithm may be more efficient than a GA based search.

In the GABH algorithm the leg position bits are first evolved by simple GA up to generation 8. All pause length bits are fixed corresponding to pause lengths of 150ms. In generation 8 GA has normally found a decent leg position pattern. From generation 9 all leg position bits are fixed. In generation 9 all possible combinations of the most significant pause length bits are tested (coarse seach) where all other bits are kept fixed. With 3 pauses in a chromosome there are 8 possible combinations of the most significant pause bits to be tested. The chromosome with the highest fitness containing the most successful most significant pause bits is kept. 8 copies of this chromosome are then made forming generation 10. In generation 10 all combinations of the next most significant pause bits are tested keeping the other bits fixed. The chromosome with the highest fitness containing the most successful next most significant pause bits are then kept. 8 copies of this chromosome are then made forming generation 11 and so on until the least significant pause bits are found in generation 14. The search is then terminated. In this way the search space given by pause lengths is searched in a coarse to fine sequence.

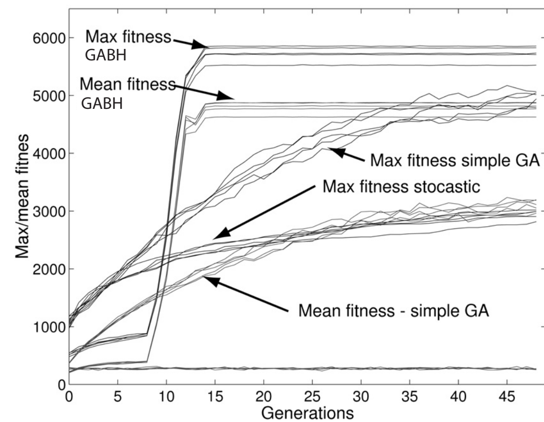

Figure 7: Comparison between simple GA, GABH and stocastic seach.

In Fig. 7 the GABH algorithm is compared to simple GA and stochastic search. As each graph represents the average fitness development over 1000 simulations, we see that the GABH algorithm is in average superior to the others in this application where the focus is fast learning in less than 20 generations. A possible objection to the proposed GABH algorithm is that heavy noise in the fitness calculations may cause the algorithm to derail and search in a non optimal region of the search space. To make the algorithm more ro-

bust an improvement could therefore be to let the algorithm run each increment over more than 1 generation and select the optimal chromosome based on fitness averaging.

4.5 Gaits obtained

The gaits obtained can be divided into three categories. Two suboptimal gaits and one optimal gait. In Fig.8-10 these gaits are illustrated. The optimal gaits were based on synchronous jumping where both legs are kicking at the same time. By kicking both feet at the same time the most power was available causing the longest jumps. Other suboptimal gaits were based on one-leg jumping or asymmetric jumping where one foot was slightly delayed with respect to the other.

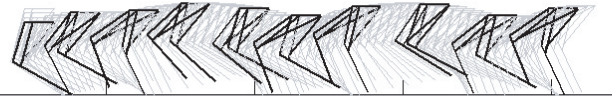

Figure 8: Suboptimal gait based on asymmetric jumping.

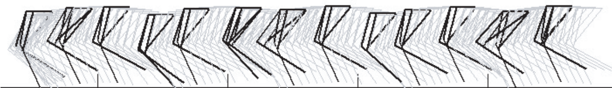

Figure 9: Suboptimal gait based on every other one-leg jumping.

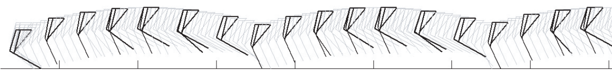

Figure 10: Optimal gaits based on synchronous jumping.

5. MEASURED RESULTS

The GABH algorithm has been tested on the pneumatic robot in an attempt to verify the theory. It was found very difficult to verify the theory accurately due to various practical side effects. One major problem was time consumption and mechanical wear out, particularly of the sandpaper shoe sole which affected the system significantly. When the robot moved, the whole system was vibrating heavily due to the quick contraction/expansion movement of the pneumatic pistons. This vibration made the robot shoe soles occasionally slip during kick-off, and this made the system very unpredictable as the robot occasionally stumbled instead of jumped even for seemingly optimal jumping patterns.

In Fig. 11 two typical fitness developments are shown for the GABH algorithm. In these examples the binary hill climbing starting point was set to the 7th generation. From the measurements we notice an improvement in fitness after this point. After the 13th generation the population was kept static, but even for repeated executions of the same chromosomes the fitness was found to vary significantly due to practical effects such as variable sole friction. However, the algorithm was found to produce proper gaits in less than 10 generations in almost all our experiments. From these few measurements it is difficult to conclude that the algorithm is working significantly better than simple GA. The only conclusion one can make so far from these measurements is that the algorithm itself is working quite well in this very noisy environment.

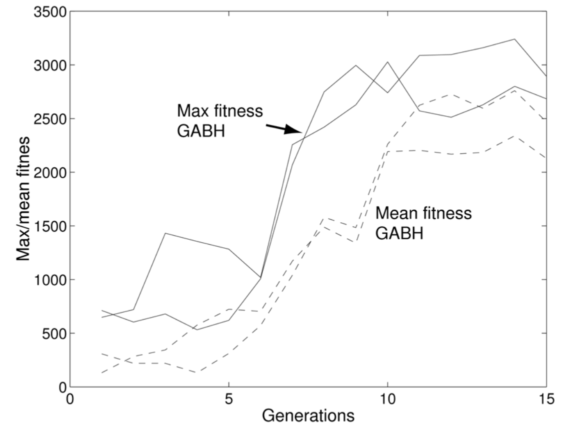

Figure 11: Measured results.

6. CONCLUSION

This paper has presented an incremental search algorithm combining GA and binary hill climbing. In various simulations this algorithm has shown to develop proper gaits significantly faster than standard GA/ES based algorithms. However, in a physical environment with practical side effects such as highly unpredictable shoe sole friction due to vibrations, varying pneumatic air pressure and wear out it has been difficult to prove in hardware that this algorithm is better than standard GA based algorithms. The algorithm itself, on the other hand was found to perform quite well in a very noisy environment.

7. REFERENCES

[1] T. Arakawa and T. Fukuda. Natural motion trajectory generation of biped locomotion robot using genetic algorithm through energy optimization. In *Proceedings of the 1996 IEEE International Conference on Systems, Man and Cybernetics*, volume 2, pages 1495–1500, 1996.

[2] E. Cantú-Paz. A survey of parallel genetic algorithms. In *Calculateurs Paralleles, Reseaux et Systems Repartis*, pages 141–171, Paris, 1998.

[3] D. Floreano and F. Mondada. Hardware solutions for evolutionary robotics. In *Proceedings of the First European Workshop on Evolutionary Robotics*, pages 137–151, London, UK, 1998. Springer-Verlag.

[4] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[5] P. C. Haddow and G. Tufte. An evolvable hardware FPGA for adaptive hardware. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, pages 553–560, California, USA, 2000. IEEE Press.

[6] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, N. Salami, N. Kajihara, and N. Otsu. Real-world applications of analog and digital evolvable hardware. *IEEE Transactions on Evolutionary Computation*, 3(3):220–235, 1999.

[7] G. Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto, and M. Fujita. Evolving robust gaits with aibo. In *ICRA*, pages 3040–3045, 2000.

[8] K. De Jong and M. A. Potter. Evolving complex structures via cooperative coevolution. In *Proceedings on the Fourth Annual Conference on Evolutionary Programming*, pages 307–317, Cambridge, MA, 1995. MIT Press.

[9] T. Kalganova. Bidirectional incremental evolution in extrinsic evolvable hardware. In *EH '00: Proceedings of the 2nd NASA/DoD workshop on Evolvable Hardware*, pages 65–74, Washington, DC, USA, 2000. IEEE Computer Society.

[10] G.B. Parker. Evolving cyclic control for a hexapod robot performing area coverage. In *Proceedings of the 2001 IEEE Computational Intelligence in Robotics and Automation*, pages 555–560, Canada, 2001.

[11] J. Torresen. A divide-and-conquer approach to evolvable hardware. In *ICES '98: Proceedings of the Second International Conference on Evolvable Systems*, pages 57–65, London, UK, 1998. Springer-Verlag.

[12] J. Torresen. An evolvable hardware tutorial. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL'2004)*, pages 821–830, Belgium, 2004.

Appendix B

Incremental Appendix

B.1 Successful Incremental Approaches

In the following appendix the prior knowledge of successful incremental approaches are documented. This chapter is also giving a short review of some of the papers handling different kinds of incremental approaches, some combined with evolutionary algorithms.

B.1.1 Bidirectional Incremental Evolution in Extrinsic Evolvable Hardware

By:Tatiana Kalganova [18].

In this paper a two-step method is proposed. The first step is to gradually decompose a complex problem using incremental evolution. The second step is to gradually make the tasks more challenging and general. The method is tested in a digital circuit domain and compared to direct evolution. The results from the experiments made, show that bidirectional incremental evolution performs significantly better than direct evolution. The process of evolution also changes when different types of decomposition are allowed. Kalganova also concludes that this approach should be applicable to many real world applications that consist of a natural hierarchy of behaviors from simple to complex.

B.1.2 Evolving both Hardware Subsystems and the Selection of Variants of Such into an Assembled System

By:Jim Tørresen [41].

I this paper, an EHW architecture for pattern classification with elements of incremental evolution has been introduced. Experiments with simple GA are tested to find the best possible combination of sub-circuits for a prosthetic hand controller circuit. There are six different motions in this prosthesis, a subsystem is evolved for each of the six motions. In the first step each of the motions are evolved separately. In the second step of the incremental approach the six motions are evolved

simultaneously. A feed-forward neural network was trained and tested with the same data sets. In 59.4 % the average performance of the EHW-architecture did better than the best case of the neural network. The author concludes that this is a promising approach for evolving systems for a complex real world application.

B.1.3 Cooperative Co-evolution: n Architecture for Evolving Co-adapted Subcomponents

By:Mitchell A. Potter and Kenneth A. De Jong [42]

To successfully apply evolutionary algorithms to the solution of increasingly complex problems, one must develop effective techniques in the form of co-evolution and co-adaptation. This paper describes an architecture for cooperative evolution of subcomponents, tested with a simple string-matching problem. The authors state that the results are quite positive in a number of ways. The first way is the architecture's ability to evolve useful decompositions as the emergence of co-operative species. Second, the cooperative co-evolutionary architecture has shown to be a general extension for any standard EC paradigm, not only for approaches like GA, and the architecture allows the authors to scale up to more complex problems than possible with standard EA. The authors also mention several other minor improvements to a current system they are exploring.

B.1.4 Exploring Knowledge Schemes for Efficient Evolution of Hardware

By: Jim Tørresen [44]

In this paper various experiments with different multipliers are presented. The multipliers used were of size 2x4-bit, 3x4-bit and 4x4-bit. They are evolved both with a traditional gate array, and with the GA-data bus architecture. For all the multipliers tested the results were up to 5-7 % better with the GA-data bus architecture. The minimum value for the GA-data bus architecture is always larger than the maximum value for the gate array architecture. The results of the multipliers didn't come out correctly. Two reasons explaining this are mentioned, the first is that the problem was quite complex, the other is that the number of experiments ran were limited. The results are concluded as an important step in the right direction. The results indicate an improvement compared to evolving with gates as building blocks. "A priori knowledge" is also an important part of this paper.

B.1.5 A Scalable Approach to Evolvable Hardware

By:Jim Tørresen [45].

In this paper various aspects of the increased complexity evolution method are investigated through a number of experiments in evolvable hardware. The scheme, called increased complexity, which contains principles from biology are applied to improve the power of evolution. The results of a character classification system

show that the total number of generations is substantially reduced by evolving subsystems. The results of a prosthetic hand controller by the use of subsystems shows that although the number of gates used increased, the total number of generations was reduced. The performance was considerably better with the method proposed.

B.1.6 Evolving Multiplier Circuits by Training Set and Training Vector Partitioning

By: Jim Tørresen [46].

In this paper experiments to evolve multiplier circuits by the use of simple GA are presented. The training sets and the vectors from the truth table are partitioned into 32 training set partitions or subsystems. The results are good, for every partition it is possible to find a correctly working subsystem in a single run. The number of generations needed to provide sufficient results with the partitions vary. The "difficult" parts of the truth table become harder to evolve with a smaller number of training set partitions. To illustrate this, an experiment with 16 training set partitions was conducted. While testing it, the results came out wrong and the number of generations needed increased a lot. Because of this Tørresen concluded that the need for small partitioning sizes was crucial to find a correctly working circuit.

B.1.7 Possibilities and Limitations of Applying Evolvable Hardware to Real-World Applications

By:Jim Tørresen [47]

This paper contains a schematic view of the important names and classifications within evolvable hardware. Tørresen also explains the different approaches and classifications. Further many publications on different real-world applications are classified and some results are discussed. To summarize, the directions for promising use of evolvable hardware when applied in online adaptable systems requiring special hardware implementations to run the application successfully, are presented. And the paper further mentions applications where an evolved circuit performs better than a traditionally designed system. Several methods for compressing the chromosome string is mentioned, one being to divide the application. This principle is based on the divide-and-conquer algorithm, it was proposed for EHW as a way of incremental evolution of the application. Further the future of EHW is outlined by mentioning the two directions where EHW will be further developed. The first is to use evolution to tune the parameters of a circuit. The second direction is to use evolution in online adaptable real-time systems. Tørresen concludes by mentioning the difficulties by using evolution in digital based applications.

B.1.8 Evolvable Reasoning Hardware: It's Application to the Genome Informatics

By:Moritoshi Yasunaga, Takahiro Tsuzuku, Kentaro Ushiyama, Ikuo Yoshihara and Jung H. Kim [48]

This article proposes a new design method called "LoDETT" (Logic Design using Evolved Truth Table). This method is used on evolvable reasoning hardware. Each task's case is transformed into truth tables. The truth tables are evolved with genetic algorithms. Circuits are synthesized from the evolved truth tables and embedded directly in the hardware. The tested hardware consists of 7 prototype Xilinx FPGA circuits. The authors conclude that the circuits perform by an accuracy of 90.5 % and are 1000 times faster than using a standard PC, Pentium 800MHz. The circuits are also quite small using few gates.

B.1.9 Evolvable Reasoning Hardware: Its Prototyping and Performance Evaluation

By:Moritoshi Yasunaga, Jung H. Kim and Ikuo Yoshihara [49]

This paper, like the previous, present the "LoDETT" design methodology, where case databases of each reasoning task are transformed into truth tables. In this paper however, the focus is to recognize unknown cases by extracting rules behind the past cases through genetic algorithms. The evolved tables are then synthesized into logic gates. Reconfigurable chips like FPGAs and CPLDs are well suited for LoDETT, because they are reconfigurable for each different task. The prototype in this matter is a Xilinx Virtex FPGA chip which is applied to the English Pronunciation Reasoning (EPR) problem. 2000 English words and pronunciations have been transformed to truth tables. The tables have been evolved with GA. The authors conclude that the average reasoning accuracy is of 82.1% and of only 270 K gates. The learning time for LoDETT using GA is a lot longer than NETTalk which finishes in 3 minutes using MY-Neupower. But, the hardware volume spent with Evolvable Reasoning Hardware (ERHW) is less than 1/50 of what MY-Neupower.

B.1.10 An Evolutionary Kernel-Based Reasoning System Using Reconfigurable VLSI: It's Hardware Prototyping and Application to the Aplicing Boundary Problem

By:Moritoshi Yasunaga, Kentaro Ushiyama, Noriyuki Aibe, Hidetoshi Fujiwara, Ikuo Yoshiyara and Jung H. Kim [50].

This paper presents a novel reasoning algorithm called Symbolical Kernel-Bases Reasoning (SKBR) The SKBR is an extension of the conventional pattern recognition algorithm Kernal-Based to the symbolical recognition and reasoning. In this paper the SKBR is applied to splicing boundary prediction problem in human DNA data in the genome informatics field. The author also shows an extracted new rule that was hidden in the consensus around the splice site boundaries. Simple GA is used

to decide where to place the "don't care-values" into the data appropriately to make kernels. The fitness evaluation is i.e how well the symbolic kernel matches the sample symbolic data. This is tested and implemented in an FPGA. Results show a short reasoning time of less than 100 micro-sec/query. The accuracy is of 95.3% for human splice site.

B.1.11 A Polygenetic, Ontogenetic and Epigenetic View of Bio-Inspired Hardware Systems

By:Morse Sipper, Eduardo Sanchez, Daniel Mange, Marco Tomassini, Andrès Pèrez-Uribe and Andrè Stauffer [51].

This paper presents the three levels of organizing life on earth. First the Polygenetic level, which concerns the temporal evolution of genetic programs like individuals and species. The second is the ontogenetic level which concerns the development process of a single multi cellular organism. The last level is the epigenetic level, which concerns the learning process during an individual organism's lifetime. This is called the POE model. The POE model can be divided into a three dimensional space where the three dimensions are partitioned into the polygenetic axis, the ontogenetic axis and the epigenetic axis. Along the polygenetic axis of bio inspired systems, also called evolvable hardware, one can find artificial evolution, LSPC's (large scale programmable circuits) and evolvable hardware. In the area of the ontogenetic axis the focus is to replicate and regenerate hardware. The main focus is following the development of a single individual, following the growth and the construction. The last axis is the epigenetic, where the learning hardware is the main subject. Learning through environmental interaction is the focus. There exist three major epigenetic systems in the living multi cellular organism, the nervous system, the immune system and the endorcine system. By presenting this POE model the authors give rise to novel systems endowed with evolutionary, reproductive, regenerative and learning capabilities.

B.1.12 Speciation as Automatic Categorical Modularization

By:Paul J. Darwen and Xin Yao [52].

In this paper an evolutionary learning system is presented. Also presented is a second approach to automatically create a repertoire of specialist strategies for a game-playing system. This relieves the human effort of deciding how to divide and specialize. The genetic algorithm speciation method used in this matter, is based on fitness sharing. It will go through a process of learning during a "Tit-for-Tat" strategy against unseen test opponents. While learning, the process automatically designs a modular system. The novelty is to use GA speciation as a modularizing mechanism. The algorithm decomposes the problem into different parts without human oversight. The tests are done by using a black box simulation with minimal prior knowledge of the learning task.

Appendix C

Media

C.1 TV performances

- **NRK1** "Schrødingers katt" January 9. 2005
  "Robotkyllingen Henriette"
- **TV2** "Klisterhjerne" October 1. 2005
- **TV Follo** October 5. 2005.
  "Robotkyllingen Henriette" - fra forskingstorget 2005

C.2 Radio performances

- **NRK P2** "Vok" December 9. 2004.
  "Robotkyllingen Henriette"
- **Kanal 24** "Superstreng" October 29. 2005
  "Superstreng - med robotforsker Lena Garder"
- **NRK Østlandssendingen** September 22. 2004
  "Knall og fall for robot-baby"

C.3 Newspaper articles

- **Verdens Gang** December 9. 2004
  "Se! En robot som lærer"
- **Aftenposten** December 7. 2004
  "Hjernen styrer håndprotese"

- **Computerworld** August 16. 2004
"Vil jobbe med robotikk"
- **Drammens Tidene** October 4. 2004
"Lærer roboter å huske"
- **Universitas** February 11. 2004
"Norske kvinner velger bort realfag"
- **Tidens Krav** December 29. 2004
"Robotene kommer!"
- **Dagsavisen** October 27. 2004
"Robotene kommer!"
- **www.forskning.no** 28. Dec. 2004
"Lager selvlærende roboter.
- **Ny teknikk** December 16. 2004
"Intelligente roboter"
- **Uniforum** May 11. 2006
"Mikroelektronikk fikk megapris"
- **www.siste.no** December 8. 2004
"Robotene kommer!"
- **www.firda.no - "Firda"** October 26. 2004
"Robotene kommer!"
- **www.forskningsradet.no "Forskning og samfunn"** September 26. 2005
"Torgsuksess i Oslo " - fra forskningstorget 2005
- **www.demokraten.no - "Demokraten"** October 26. 2004
"Robotene kommer!"
- **www.hadeland.net - "Hadeland"** October 26. 2004
"Robotene kommer!"
- **www.rb.no - "Romerikes Blad"** October 26. 2004
"Robotene kommer!"
- **www.sognavis.no - "Sogn Avis"** October 26. 2004
"Robotene kommer!"
- **www.tvedestrandsposten.no - "Tvedestrandsposten"** October 26. 2004.
"Robotene kommer!"
- **www.lofotposten.no - "Lofotposten"** October 26. 2004
"Robotene kommer!"

C.4 Magazine articles

- **Apollon** (science magazine) December 6. 2004
  "Lager selvlærende roboter"

- **Elektronikk** August 7. 2005
  "Dagens roboter prøver ut genetiske søkealgoritmer".

C.5 Invited Talk

- **Invited talk** Intervensjonssenteret - Rikshospitalet February 11. 2005
  "Robotkyllingen Henriette / Humanoide roboter, status for 2004"

Appendix D

Dictionary

This appendix of definitions is not written in a scientific matter, there are no references of these explanations. The majority of words are explained within the thesis, with references. This dictionary is meant for quick and easy consulting.

- **Algorithm / Routine**
  Part of a computer program performing a certain task
- **A priori Knowledge**
  Knowledge or instincts given to the robot prior to evaluation, to avoid certain behavior.
- **Binary Hillclimbing**
  A search method that always chooses the steepest direction in a search space, i.e. the best solution seen from the situated point.
- **Chromosome**
  Solution, represented by a bit string containing the information to control the robot.
- **Crossover**
  Genetic operator taking two parent chromosomes, divides them and put them together making two new (children) chromosomes.
- **Cyclic Evolution**
  Chromosomes are repeated in cycles, inspired by actual gaits.
- **Cylinder bits**
  Parts of the chromosome that controls the cylinders.
- **E.g.**
  "For instance".
- **Elitism**
  The two best chromosomes in a population that are directly transferred to the next generation. Also called clone.

- **Evolutionary Strategy**
  The population is ranked and the best chromosome is copied up into 40 percent of the next generation, 30 percent of the next best chromosome, 20 percent of the third best chromosome and 10 percent of the fourth best chromosome are also chosen to proceed to the next generation.

- **Fitness-based selection**
  Chooses the chromosomes in a generation that have the highest fitness score to perform in the next generation.

- **Fitness Landscape**
  See search space, can be of different shapes and structures.

- **Fitness measure**
  Feedback from the robot/simulator that is handled by the fitness routine. Fitness score, see fitness routine.

- **Fitness Routine**
  A routine that receives feedback from the robot/simulator and translates the feedback into a measure or score telling how well the robot behaved.

- **GABH algorithm**
  The novel algorithm described in chapter 5. Starting with cylinder bit evaluation, with fixed pauses, for 10 generations. Then the optimal sequence of pause bits are found by using a binary hill climb type of search.

- **Gait**
  Walking pattern, way to move legs to go forward.

- **Generation**
  One collection of solutions/chromosomes tested in the search space.

- **Generational GA**
  GA where the entire population is updated every generation.

- **Genetic Operators**
  Rules making diversity of chromosomes in a population, E.g. crossover, mutation and selection.

- **I.e.**
  "That means".

- **Incremental GA**
  Divide the search space into several parts that evolve separately.

- **Incremental Pause Tree**
  An algorithm evaluating the cylinder bits prior to the pause bits. After a certain generation the pauses are evaluated and the pause lengths are chosen

from a tree structure. There are variations within the pause tree methods, some have cylinder evaluation inside the pause tree evaluation as well.

- **Mutation**
  Genetic operator that takes one bit and changes the value to the opposite.

- **Online Learning**
  Machine Learning performed on real robots, not simulation, and without human interaction within the evolution.

- **Pause bits**
  The parts of the chromosome that control the pause lengths.

- **Population**
  Collection of solutions/chromosomes.

- **Rank-based selection**
  All the chromosomes are ranked and given a new fitness value according to their rank. Still they have to be chosen by another selection method.

- **Roulette wheel Selection**
  Each chromosome in a generation is given a certain amount of a roulette wheel according to it's fitness value. The chromosomes chosen to proceed to the next generation are selected by spinning the wheel and randomly choosing chromosomes.

- **Search Space**
  Room/space full of solutions, some are better than others.

- **Selection schemes**
  Different ways of reproduction of chromosomes, E.g. fitness based selection, Roulette wheel selection, Rank based selection, Tournament selection and Evolutionary strategy.

- **Simulator**
  Computer program that behaves like the robot and gives feedback like the robot.

- **Steady-state GA**
  GA where only parts of the population are updated every generation.

- **Tournament selection**
  Chooses two chromosomes randomly and the chromosome with the highest fitness score is chosen to proceed to the next generation.

Appendix E

Source Code

This appendix includes the source code of the program running on the *MES Robotic Chicken*.

```
#include "stdio.h"
#include <stdlib.h>
#include "windows.h"
#include <time.h>
#include <math.h>
#include <iostream>

#define IDV   10    // individ
#define GEN   3     // gen
#define BIT   8     // antall bit (4 sylinder-bit og 4 pause-bit)

#define SMUT  25    // antall muteringer skjer per populasjon
#define ELLI  2     // ELLI == '0': ingen ellitisme || '2': Ellitisme
#define COX   0     // Antall individ som skal krysses (rundes ned ved oddetall)
#define SM    2     // 1 = Roulette wheel 2 = Evolutionary Strategie 3 = Tournament

//#define PAUSE 0       // 0 = fast pause, må settes. 1 = Pausen som en del av GA, 2 = Pausetre 3 = ellers

#define REP   3     // antall repetisjoner av chromosomet før fitness leses av
#define GNR   50    // antall generasjoner
#define POPS  100   // antall kjøringer som midles

#define TYPE  1     // 1 = stokastisk søk, 2 = Pausetre, 3 = GA, 4 = TEST-position, 5 = Fast Pause 6 = GABH
#define GB    2     // 2 = pausetre-funksjon (MÅ STÅ på 3 UNDER GA)
#define PI    3.14152
```

```
/***** Pre deklarering av funksjoner. *****/

int     bin2int(int[12]);
void    confPort_0_12_in(void);
void    confPort_0_7_out(void);
int     ellitisme(int);
int     getBit(int, int);
int     gray2bin(int);
int     *int2bin(int);
void    kjor(int);
void    kryss(void);
int     maxi(void);
void    muter();
int     random(int);
int     rettVerdi(int);
int     rulett(void);
int     rank(void);
void    pre();
void    print2file(int);
void    read4file();
int     snitt(void);
int     matti(void);
int     bin2grayGenerell(int);
int     es(void);
int     tournament(void);
int     gabh(void);

int runC(double pau, int C1, int C2, int C3, int C4, int Res, int SkrivUt, FILE* filPek);
double cyl1_lng(int C, double incr);
double cyl2_lng(int C, double incr);
double cyl3_lng(int C, double incr);
double cyl4_lng(int C, double incr);

int    likeBevegelser(double L1_old,double L2_old,double L3_old,double L4_old,int C1,int C2,int C3,int C4);
void   fotlengder(double* X1, double* Y1, double* X2, double* Y2, double Sx, double Sy, double L1, double L2, double L3, double I

//**********************************************************************************************
// runC : kjører ut ett gen (en posisjon med tilhørende pause)
// returnerer chick posisjon i millimeter etter utført gen (statisk int variabel),
// distansen øker helt til simulatoren blir resatt (chick går ikke i ring)
// pau = pausetid (sek), Cn er 0/1 (cylinder nr.n inn/ut)
// Res = 1 : simulatoren blir resatt (må gjøres før første kjøring)
// Res = 0 : ordinær simulering uten resett
// SkrivUt = 0 : skriver ikke kroppsposisjoner til fil under simulering
// SkrivUt = 1 : skriver fotposisjoner til fil under simulering format (int'er) (millimeter)
// SkrivUt = 2 : skriver kropsposisjoner til fil under simulering format (int'er) (millimeter)
//**********************************************************************************************

/***** Globale deklareringer. *****/

int     hovedarray[IDV][GEN];
int     temparray [IDV][GEN];
int     konstarray[IDV][GEN];
int     fitness   [IDV];        // Fitnessarray
int     fit2      [IDV];
int     bin1      [12];         // Array for de binære tallene som skal "kjøre" roboten
int     gray      [8];          // Array for å hente inn gray-kodingen.
int     elli      [GEN*2];
int     q         [256];
int     XV = 0;
int     MTALL = 0;
int     MEAN = 0;
int     bix = 0;
int     ra = 0;

void main()
{

    int     a, a1, a2, a3, b, c, e, i=0, f, g, x, pil, *p, *p2, tre=0, to=0, div = 0, tee;
    double  tid = 0.05, psum = 0.0;
    int     sjekk=0, fart=0, teller = 0;
    int     cc[GEN];
    bool    t;
  // read4file(); // kommando som kjøres etterpå for å lese inn filene og ta mean av de 5 * 100 kjøringene.
/**/

    for(bix=0; bix<5; bix++){ printf("\n ** BIX: %2d ** \n\n", bix); for(i=0; i<POPS; i++) // #grafer / midlinger
    {
        MTALL = 0; MEAN = 0;                    // nullstiller
        for(c=0; c<IDV; c++)fitness[c] = 0;     // nullstiller
        pre();                                  // initializing routine

        for(x=0; x<GNR; x++)                    // generasjoner
        {
            for(c=0; c<IDV; c++){ fitness[c] = 0; q[c] = 0;}    // nullstiller

            for(a=0; a<IDV; a++)
            {
                fitness[a] = 0;
                tid = 0.05; b = runC(tid, 0, 0, 0, 0, 1 ,0, 0); // setter vinkelen til en fast standard

                if(TYPE == 1)
                { // STOKASTISK SØK
                    if(GB == 1){ //  tester på om randomtallet har vært brukt tidligere
                        t = TRUE;
                        while(t == TRUE){
                            c = random(255);
                            if(q[c] == 0){
```

```
                    q[c] = 1;
                    //hovedarray[a][b] = c;
                    cc[0] = random(255); cc[1] = random(255); cc[2] = random(255);
                    t = FALSE;
                }   // end if
            }   // end while
        }else if(GB == 2){ // gjenbruk er lov
            cc[0] = random(255); cc[1] = random(255); cc[2] = random(255);
            psum = 0.0;
        }   // end if-else
        for(e=0; e<GEN; e++)
        {
            tid = 0.05;
            p = int2bin(cc[0]);

            if(p[4] == 1) tid=tid+0.0375;
            if(p[5] == 1) tid=tid+0.075;
            if(p[6] == 1) tid=tid+0.15;
            if(p[7] == 1) tid=tid+0.30;

            runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            psum = psum + tid; tid = 0.05;
            p = int2bin(cc[1]);

            if(p[4] == 1) tid=tid+0.0375;
            if(p[5] == 1) tid=tid+0.075;
            if(p[6] == 1) tid=tid+0.15;
            if(p[7] == 1) tid=tid+0.30;

            runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            psum = psum + tid; tid = 0.05;
            p = int2bin(cc[2]);

            if(p[4] == 1) tid=tid+0.0375;
            if(p[5] == 1) tid=tid+0.075;
            if(p[6] == 1) tid=tid+0.15;
            if(p[7] == 1) tid=tid+0.30;

            fitness[a] = fitness[a] + runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            psum = psum + tid;

        }   //end for

        fart = fitness[a] / psum;
        fart = fart * (fart>0);
        fitness[a] = fart;

    }else if(TYPE == 2)
    { // PAUSETRE-FUNKSJONEN
        psum = 0.0;
        for(b=0; b<GEN; b++)
        {
            if(x==9)konstarray[a][b]=hovedarray[a][b]; // dersom du er i generasjon 9, spar denne generasjonen i kons
            if(x==29)konstarray[a][b]=hovedarray[a][b];
            tid = 0.05;
            p = int2bin(hovedarray[a][0]);
            if(x<10){    tid = 0.1;                      }
            if(x>20){    if(p[4] == 1) tid=tid+0.0375;   }
            if(x>15){    if(p[5] == 1) tid=tid+0.075;    }
            if(x>10){    if(p[6] == 1) tid=tid+0.15;
                         if(p[7] == 1) tid=tid+0.30;     }

            if((x>10 && x<19) || (x>29 && x<39))div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            else{
                p = int2bin(konstarray[a][0]);
                div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            }

            psum = psum + tid; tid = 0.05;
            p = int2bin(hovedarray[a][1]);

            if(GNR<10){ tid = 0.1;                       }
            if(GNR>20){ if(p[4] == 1) tid=tid+0.0375;    }
            if(GNR>15){ if(p[5] == 1) tid=tid+0.075;     }
            if(GNR>10){ if(p[6] == 1) tid=tid+0.15;
                        if(p[7] == 1) tid=tid+0.30;      }

            if((x>10 && x<19) || (x>29 && x<39))div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            else{
                p = int2bin(konstarray[a][1]);
                div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            }

            psum = psum + tid; tid = 0.05;
            p = int2bin(hovedarray[a][2]);

            if(GNR<10){ tid = 0.1;                       }
            if(GNR>20){ if(p[4] == 1) tid=tid+0.0375;    }
            if(GNR>15){ if(p[5] == 1) tid=tid+0.075;     }
            if(GNR>10){ if(p[6] == 1) tid=tid+0.15;
                        if(p[7] == 1) tid=tid+0.30;      }

            if((x>10 && x<19) || (x>29 && x<39))div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            else{
                p = int2bin(konstarray[a][2]);
                div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            }
```

```
            fitness[a] = fitness[a] + div;
            psum = psum + tid;

        }   // end for

        fart = fitness[a] / psum;
        fart = fart * (fart>0);
        fitness[a] = fart;

    }else if(TYPE == 3)
    {// GA-KJØRING

        psum = 0.0;
        for(b=0; b<GEN; b++)
        {
            tid = 0.05;
            p = int2bin(hovedarray[a][0]);

            if(p[4] == 1) tid=tid+0.0375;
            if(p[5] == 1) tid=tid+0.075;
            if(p[6] == 1) tid=tid+0.15;
            if(p[7] == 1) tid=tid+0.30;

            div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            psum = psum + tid; tid = 0.05;
            p = int2bin(hovedarray[a][1]);

            if(p[4] == 1) tid=tid+0.0375;
            if(p[5] == 1) tid=tid+0.075;
            if(p[6] == 1) tid=tid+0.15;
            if(p[7] == 1) tid=tid+0.30;

            div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            psum = psum + tid; tid = 0.05;
            p = int2bin(hovedarray[a][2]);

            if(p[4] == 1) tid=tid+0.0375;
            if(p[5] == 1) tid=tid+0.075;
            if(p[6] == 1) tid=tid+0.15;
            if(p[7] == 1) tid=tid+0.30;

            div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
            fitness[a] = fitness[a] + div;
            psum = psum + tid;

        }   // end for

        fart = fitness[a] / psum;
        fart = fart * (fart>0);
        fitness[a] = fart;
                            // end GA-kjøring
    }else if(TYPE == 4)
    {
        tid = 0.1;
        b = runC(tid, 1, 0, 0, 1, 0 ,0, 0); printf("Run: %d \n", a);
        b = runC(tid, 1, 1, 0, 0, 0 ,0, 0); printf("Run: %d \n", a);

    }else if(TYPE == 5)
    {   // Fast pause

        tid = 0.1; psum = 0.0;
        for(b=0; b<GEN; b++)
        {
            p = int2bin(hovedarray[a][0]); div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0); psum = psum + tid;
            p = int2bin(hovedarray[a][1]); div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0); psum = psum + tid;
            p = int2bin(hovedarray[a][2]); div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0); psum = psum + tid;
            fitness[a] = fitness[a] + div;
        }   // end for

        fart = fitness[a] / psum;
        fart = fart * (fart>0);
        fitness[a] = fart;

    }else if(TYPE == 6){ // GABH-algoritmen

        psum = 0.0;
        if(x<10){

            tid = 0.1; // Optimalt = 0.1
            for(b=0; b<GEN; b++)
            {
                p = int2bin(hovedarray[a][0]);
                div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
                psum = psum + tid;

                p = int2bin(hovedarray[a][1]);
                div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
                psum = psum + tid;

                p = int2bin(hovedarray[a][2]);
                div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
                psum = psum + tid;

                fitness[a] = fitness[a] + div;
            }   // end for
```

```
            fart = fitness[a] / psum;
            fart = fart * (fart>0);
            fitness[a] = fart;

        }else if(x>9){  // KONSTARRAY SOM SKAL BRUKES

            p2 = int2bin(a);
            a1 = p2[0]; a2 = p2[1]; a3 = p2[2];

            for(b=0; b<GEN; b++) // gjentagelsene (3 rep)
            {
                tid = 0.05;
                p=int2bin(konstarray[a][0]);
                if(a1==1){
                    if(x==10){p[4]=1;}
                    if(x==11){p[5]=1;}
                    if(x==12){p[6]=1;}
                    if(x==13){p[7]=1;}
                }else{
                    if(x==10){p[4]=0;}
                    if(x==11){p[5]=0;}
                    if(x==12){p[6]=0;}
                    if(x==13){p[7]=0;}
                }

                 if(x==10){ p[5]=0; p[6]=0; p[7]=0; }
            else if(x==11){         p[6]=0; p[7]=0; }
            else if(x==12){                 p[7]=0; }

            konstarray[a][0] = bin2int(p);

                if(p[7]==1)tid = tid + 0.0375;
                if(p[6]==1)tid = tid + 0.075;
                if(p[5]==1)tid = tid + 0.15;
                if(p[4]==1)tid = tid + 0.3;

                div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
                psum = psum + tid;

                tid = 0.05;
                p=int2bin(konstarray[a][1]);
                if(a2==1){
                    if(x==10){p[4]=1;}
                    if(x==11){p[5]=1;}
                    if(x==12){p[6]=1;}
                    if(x==13){p[7]=1;}
                }else{
                    if(x==10){p[4]=0;}
                    if(x==11){p[5]=0;}
                    if(x==12){p[6]=0;}
                    if(x==13){p[7]=0;}
                }

                     if(x==10){ p[5]=0; p[6]=0; p[7]=0; }
                else if(x==11){         p[6]=0; p[7]=0; }
                else if(x==12){                 p[7]=0; }

                konstarray[a][1] = bin2int(p);

                if(p[7]==1)tid = tid + 0.0375;
                if(p[6]==1)tid = tid + 0.075;
                if(p[5]==1)tid = tid + 0.15;
                if(p[4]==1)tid = tid + 0.3;

                div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
                psum = psum + tid;

                tid = 0.05;
                p=int2bin(konstarray[a][2]);
                if(a3==1){
                    if(x==10){p[4]=1;}
                    if(x==11){p[5]=1;}
                    if(x==12){p[6]=1;}
                    if(x==13){p[7]=1;}
                }else{
                    if(x==10){p[4]=0;}
                    if(x==11){p[5]=0;}
                    if(x==12){p[6]=0;}
                    if(x==13){p[7]=0;}}

                     if(x==10){ p[5]=0; p[6]=0; p[7]=0; }
                else if(x==11){         p[6]=0; p[7]=0; }
                else if(x==12){                 p[7]=0; }

                konstarray[a][2] = bin2int(p);

                if(p[7]==1)tid = tid + 0.0375;
                if(p[6]==1)tid = tid + 0.075;
                if(p[5]==1)tid = tid + 0.15;
                if(p[4]==1)tid = tid + 0.3;
                div = runC(tid, p[0], p[1], p[2], p[3], 0, 0, 0);
                psum = psum + tid;

                fitness[a] = fitness[a] + div;

            }  // end for
            fart = fitness[a] / psum;
```

```
                        fart = fart * (fart>0);
                        fitness[a] = fart;
                    }   // end gnr-greia

                }       // end TYPE
                print2file(2);
            } // end IDV

            print2file(3);      // skriver til statistikk-filene

            if(TYPE == 2 || TYPE == 3 || TYPE == 5 || (TYPE == 6)){ // GA

                if(TYPE == 6 && x>=9){ // KUN GAHB

                    ra = rank(); // ra vil inneholde index'en til den beste

                    // seleksjonen: Legger inn det beste individet i hele pop'en
                    for(a=0; a<IDV; a++){
                        if(x>9){
                            konstarray[a][0] = konstarray[ra][0];
                            konstarray[a][1] = konstarray[ra][1];
                            konstarray[a][2] = konstarray[ra][2];
                        }if(x==9){
                            konstarray[a][0] = hovedarray[ra][0];
                            konstarray[a][1] = hovedarray[ra][1];
                            konstarray[a][2] = hovedarray[ra][2];
                        }   // end if           int hovedarray[][1]
                    }   // end for
                }else{  // end GABH-behandling

                    ellitisme(1);       // kopierer de 2 beste individene inn i elli-arrayene.
                                        // Alt ligger nå i hovedarray

                    // Seleksjons-modeller:
                    ra = rank();        // rangerer og gir ny fitness basert på rank
                    if(SM == 1) pil = rulett(); // Alt ligger nå i temp-arrayet
                    if(SM == 2) es();
                    if(SM == 3) tournament();

                    kryss();            // krysser (med innebygget muteringsfunksjon
                                        // alt ligger nå i hovedarrayet
                    muter();

                    ellitisme(2);       // Legger de to beste individene overskrive 2 første i hovedarray
                }
            }   // end if
        }   // end GNR

        print2file(4);

    }   // end POPS
    }   // end bix

    /**/
    std::cout<< '\n'<<"Programmet avsluttes..."<< '\n';

} // end main



void pre() // fyller opp utgangspunkt arrayet med tall opp til 16 (pga 4 bit).
{
    int a, b, d=0;

    for(a=0; a<IDV; a++)
    {
        for(b=0; b<GEN; b++)
        {
            hovedarray[a][b] = random(255);
        }
    }
}   // end function

//*************************************** PRINTE FUNKSJONER *****************************************

void print2file(int c)
{
    FILE   *fp1, *fp2, *fp3, *fp4;

    int     sjekk = 0, snittall=0, maxtall=0, s=0;
    if(c==1){ // åpner for kyllingfila
             // fp1 = fopen("../../../../../MATLAB6p1/work/chicki.m", "w");
             // if(fp1 == NULL) printf("Feil under åpning av fil A\n");

    }else if(c==2){

        fp2 = fopen("../../../../../MATLAB6p1/work/GOD1.m", "a");

        if(fp2 == NULL)printf("Feil i filåpning.\n");

        snittall = snitt(); if(snittall > MEAN) MEAN = snittall;
        maxtall  = maxi();  if(maxtall > MTALL) MTALL = maxtall; // passer på å hele tiden ha de største max- og mean-tallene

        fprintf(fp2, "%d %d", MTALL, MEAN);

        sjekk = fclose(fp2);
        if(sjekk != 0) std::cout<<"Fitness filen ble ikke lukket ordentlig."<<'\n';
```

```
    }else if(c==3) {

        if(bix == 0){
            fp3 = fopen("../../../../../MATLAB6p1/work/POPmax1.m", "a");
            fp4 = fopen("../../../../../MATLAB6p1/work/POPmean1.m", "a");
        }else if(bix == 1){
            fp3 = fopen("../../../../../MATLAB6p1/work/POPmax2.m", "a");
            fp4 = fopen("../../../../../MATLAB6p1/work/POPmean2.m", "a");
        }else if(bix == 2){
            fp3 = fopen("../../../../../MATLAB6p1/work/POPmax3.m", "a");
            fp4 = fopen("../../../../../MATLAB6p1/work/POPmean3.m", "a");
        }else if(bix == 3){
            fp3 = fopen("../../../../../MATLAB6p1/work/POPmax4.m", "a");
            fp4 = fopen("../../../../../MATLAB6p1/work/POPmean4.m", "a");
        }else if(bix == 4){
            fp3 = fopen("../../../../../MATLAB6p1/work/POPmax5.m", "a");
            fp4 = fopen("../../../../../MATLAB6p1/work/POPmean5.m", "a");
        }

        if(fp3 == NULL || fp4 == NULL) printf("\nFEIL i filåpning 3 og 4 !\n");

        fprintf(fp3, "%d ", MTALL); fprintf(fp4, "%d ", MEAN);

        sjekk = fclose(fp3) + fclose(fp4);
        if(sjekk != 0) std::cout<<"Filene ble ikke lukket ordentlig."<<'\n';

    }else if(c==4) {

        if(bix == 0){
            fp3 = fopen("../../../../../MATLAB6p1/work/POPmax1.m", "a");
            fp4 = fopen("../../../../../MATLAB6p1/work/POPmean1.m", "a");
        }else if(bix == 1){
            fp3 = fopen("../../../../../MATLAB6p1/work/POPmax2.m", "a");
            fp4 = fopen("../../../../../MATLAB6p1/work/POPmean2.m", "a");
        }else if(bix == 2){
            fp3 = fopen("../../../../../MATLAB6p1/work/POPmax3.m", "a");
            fp4 = fopen("../../../../../MATLAB6p1/work/POPmean3.m", "a");
        }else if(bix == 3){
            fp3 = fopen("../../../../../MATLAB6p1/work/POPmax4.m", "a");
            fp4 = fopen("../../../../../MATLAB6p1/work/POPmean4.m", "a");
        }else if(bix == 4){
            fp3 = fopen("../../../../../MATLAB6p1/work/POPmax5.m", "a");
            fp4 = fopen("../../../../../MATLAB6p1/work/POPmean5.m", "a");
        }

        if(fp3 == NULL || fp4 == NULL)printf("feil under åpning av fil A \n");

        fprintf(fp3, "\n"); fprintf(fp4, "\n");


        sjekk = fclose(fp3) + fclose(fp4);
        if(sjekk != 0) std::cout<<"Filene ble ikke lukket ordentlig."<<'\n';

        MTALL = 0; MEAN = 0;
        snittall=0; maxtall=0;

    }   // end if-else
}   // end function


void read4file() // leser fra filen og plotter grafer av filene produsert i write2file()
{ // 50 array fordi det er 5 * antall ganger.
    FILE    *fp5, *fp6, *fp7;
    int     g1[POPS], g2[POPS], g3[POPS], g4[POPS], g5[POPS], g6[POPS], g7[POPS], g8[POPS], g9[POPS], g10[POPS];
    int     g11[POPS], g12[POPS], g13[POPS], g14[POPS], g15[POPS], g16[POPS], g17[POPS], g18[POPS], g19[POPS], g20[POPS];
    int     g21[POPS], g22[POPS], g23[POPS], g24[POPS], g25[POPS], g26[POPS], g27[POPS], g28[POPS], g29[POPS], g30[POPS];
    int     g31[POPS], g32[POPS], g33[POPS], g34[POPS], g35[POPS], g36[POPS], g37[POPS], g38[POPS], g39[POPS], g40[POPS];
    int     g41[POPS], g42[POPS], g43[POPS], g44[POPS], g45[POPS], g46[POPS], g47[POPS], g48[POPS], g49[POPS], g50[POPS];
    int     a=0, tall=0, i, s, sjekk = 0, sjekk2 = 0;

    for(s=1; s<=10; s++){
        printf("\n\n");

        if(s==1){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMax1.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmax1.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        }else if(s==2){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMax2.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmax2.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        }else if(s==3){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMax3.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmax3.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        }else if(s==4){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMax4.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmax4.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        }else if(s==5){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMax5.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmax5.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        }else if(s==6){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMean1.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmean1.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
```

```
        sjekk = fclose(fp3) + fclose(fp4);
        if(sjekk != 0) std::cout<<"Filene ble ikke lukket ordentlig."<<'\n';

        MTALL = 0; MEAN = 0;
        snittall=0; maxtall=0;

    }   // end if-else
}   // end function



void read4file() // leser fra filen og plotter grafer av filene produsert i write2file()
{ // 50 array fordi det er 5 * antall ganger.
    FILE    *fp5, *fp6, *fp7;
    int     g1[POPS], g2[POPS], g3[POPS], g4[POPS], g5[POPS], g6[POPS], g7[POPS], g8[POPS], g9[POPS], g10[POPS];
    int     g11[POPS], g12[POPS], g13[POPS], g14[POPS], g15[POPS], g16[POPS], g17[POPS], g18[POPS], g19[POPS], g20[POPS];
    int     g21[POPS], g22[POPS], g23[POPS], g24[POPS], g25[POPS], g26[POPS], g27[POPS], g28[POPS], g29[POPS], g30[POPS];
    int     g31[POPS], g32[POPS], g33[POPS], g34[POPS], g35[POPS], g36[POPS], g37[POPS], g38[POPS], g39[POPS], g40[POPS];
    int     g41[POPS], g42[POPS], g43[POPS], g44[POPS], g45[POPS], g46[POPS], g47[POPS], g48[POPS], g49[POPS], g50[POPS];
    int     a=0, tall=0, i, s, sjekk = 0, sjekk2 = 0;

    for(s=1; s<=10; s++){
        printf("\n\n");

        if(s==1){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMax1.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmax1.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        }else if(s==2){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMax2.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmax2.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        }else if(s==3){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMax3.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmax3.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        }else if(s==4){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMax4.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmax4.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        }else if(s==5){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMax5.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmax5.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        }else if(s==6){
            fp6     = fopen("../../../../../MATLAB6p1/work/GrafMean1.m", "w");
            fp5     = fopen("../../../../../MATLAB6p1/work/POPmean1.m", "r");
            if(fp5 == NULL || fp6 == NULL)printf("selvmord! %d\n", s);
        fscanf(fp5," %d", &i);   g30[a] = i; //printf(" %d ", i);
        }if(GNR > 30){
        fscanf(fp5," %d", &i);   g31[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g32[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g33[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g34[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g35[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g36[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g37[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g38[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g39[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g40[a] = i; //printf(" %d ", i);
        }if(GNR > 40){
        fscanf(fp5," %d", &i);   g41[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g42[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g43[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g44[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g45[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g46[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g47[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g48[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g49[a] = i; //printf(" %d ", i);
        fscanf(fp5," %d", &i);   g50[a] = i; //printf(" %d ", i);
        }

        for(a=1; a<POPS; a++){
            //printf(". ");
            fscanf(fp5," %d", &i); g1[a] = g1[a-1] + i;
            fscanf(fp5," %d", &i); g2[a] = g2[a-1] + i;
            fscanf(fp5," %d", &i); g3[a] = g3[a-1] + i;
            fscanf(fp5," %d", &i); g4[a] = g4[a-1] + i;
            fscanf(fp5," %d", &i); g5[a] = g5[a-1] + i;
            fscanf(fp5," %d", &i); g6[a] = g6[a-1] + i;
            fscanf(fp5," %d", &i); g7[a] = g7[a-1] + i;
            fscanf(fp5," %d", &i); g8[a] = g8[a-1] + i;
            fscanf(fp5," %d", &i); g9[a] = g9[a-1] + i;
            fscanf(fp5," %d", &i); g10[a] = g10[a-1] + i;
            if(GNR > 10){
            fscanf(fp5," %d", &i); g11[a] = g11[a-1] + i;
            fscanf(fp5," %d", &i); g12[a] = g12[a-1] + i;
            fscanf(fp5," %d", &i); g13[a] = g13[a-1] + i;
            }if(GNR > 13){
            fscanf(fp5," %d", &i); g14[a] = g14[a-1] + i;
            fscanf(fp5," %d", &i); g15[a] = g15[a-1] + i;
            fscanf(fp5," %d", &i); g16[a] = g16[a-1] + i;
            fscanf(fp5," %d", &i); g17[a] = g17[a-1] + i;
            fscanf(fp5," %d", &i); g18[a] = g18[a-1] + i;
            fscanf(fp5," %d", &i); g19[a] = g19[a-1] + i;
            fscanf(fp5," %d", &i); g20[a] = g20[a-1] + i;
            }if(GNR > 20){
```

```
            fscanf(fp5," %d", &i); g21[a] = g21[a-1] + i;
            fscanf(fp5," %d", &i); g22[a] = g22[a-1] + i;
            fscanf(fp5," %d", &i); g23[a] = g23[a-1] + i;
            fscanf(fp5," %d", &i); g24[a] = g24[a-1] + i;
            fscanf(fp5," %d", &i); g25[a] = g25[a-1] + i;
            fscanf(fp5," %d", &i); g26[a] = g26[a-1] + i;
            fscanf(fp5," %d", &i); g27[a] = g27[a-1] + i;
            fscanf(fp5," %d", &i); g28[a] = g28[a-1] + i;
            fscanf(fp5," %d", &i); g29[a] = g29[a-1] + i;
            fscanf(fp5," %d", &i); g30[a] = g30[a-1] + i;
            }if(GNR > 30){
            fscanf(fp5," %d", &i); g31[a] = g31[a-1] + i;
            fscanf(fp5," %d", &i); g32[a] = g32[a-1] + i;
            fscanf(fp5," %d", &i); g33[a] = g33[a-1] + i;
            fscanf(fp5," %d", &i); g34[a] = g34[a-1] + i;
            fscanf(fp5," %d", &i); g35[a] = g35[a-1] + i;
            fscanf(fp5," %d", &i); g36[a] = g36[a-1] + i;
            fscanf(fp5," %d", &i); g37[a] = g37[a-1] + i;
            fscanf(fp5," %d", &i); g38[a] = g38[a-1] + i;
            fscanf(fp5," %d", &i); g39[a] = g39[a-1] + i;
            fscanf(fp5," %d", &i); g40[a] = g40[a-1] + i;
            }if(GNR > 40){
            fscanf(fp5," %d", &i); g41[a] = g41[a-1] + i;
            fscanf(fp5," %d", &i); g42[a] = g42[a-1] + i;
            fscanf(fp5," %d", &i); g43[a] = g43[a-1] + i;
            fscanf(fp5," %d", &i); g44[a] = g44[a-1] + i;
            fscanf(fp5," %d", &i); g45[a] = g45[a-1] + i;
            fscanf(fp5," %d", &i); g46[a] = g46[a-1] + i;
            fscanf(fp5," %d", &i); g47[a] = g47[a-1] + i;
            fscanf(fp5," %d", &i); g48[a] = g48[a-1] + i;
            fscanf(fp5," %d", &i); g49[a] = g49[a-1] + i;
            fscanf(fp5," %d", &i); g50[a] = g50[a-1] + i;
        }}

        if(s == 2){fclose(fp6); fp6 = fopen("../../../../../MATLAB6p1/work/GrafMax2.m", "w"); }

        tall = g1[POPS-1]/POPS;  fprintf(fp6,"%d ", tall); printf("%d ", tall);
        tall = g2[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g3[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g4[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g5[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g6[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g7[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g8[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g9[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g10[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        if(GNR > 10){
        tall = g11[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g12[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g13[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        }if(GNR > 13){
        tall = g14[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g15[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g16[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g17[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g18[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g19[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g20[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        }if(GNR > 20){
        tall = g21[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g22[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g23[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g24[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g25[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g26[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g27[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g28[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g29[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g30[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        }if(GNR > 30){
        tall = g31[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g32[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g33[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g34[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g35[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g36[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g37[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g38[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g39[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g40[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        }if(GNR > 40){
        tall = g41[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g42[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g43[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g44[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g45[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g46[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g47[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g48[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g49[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);
        tall = g50[POPS-1]/POPS;  fprintf(fp6,"%d ", tall);

        }
        sjekk = fclose(fp5);
        if(sjekk != 0) printf("Filene ble ikke lukket ordentlig.\n");
        sjekk = 0;
        sjekk = fclose(fp6);
        if(sjekk != 0) printf("problems ved lukking av fil!");

    }   // end for
}   // end routine
```

```
/***** FITNESS-FUNKSJONER *****/


int snitt()
{
    int mean = 0, a;

    for(a=0; a<IDV; a++){ mean = mean + fitness[a]; }   // legger sammen alle fitnessene
    mean = mean/IDV;                                    // deler på # fitness
    if(mean > MEAN) MEAN = mean;                        // sjekker om den er større enn tidl.

    return mean;
}


int maxi()
{
    int big = 0, a;

    for(a=0; a<IDV; a++){ if(fitness[a]>big) big = fitness[a]; } // finner den største i fitness-arrayet.

    return big;
}


/********** BEHANDLINGS RUTINER  **********/


int getBit(int numb, int index)
{
    return((numb & (int) pow(2,index))>0);
}



int rank()
{
    int a, b=1, c, e=0, num=0,t=0, s=0;
    int n[IDV];

    for(a=0; a<IDV; a++)n[a]=0; // nullstiling av n-array

    for(e=0; e<IDV; e++){
        num = 0;
        for(a=0; a<IDV; a++){

            if(n[a] == 0 && fitness[a] >= num){
                c = a;
                num = fitness[a];
            }   // end if
        }   // end for
        n[c] = 1;
        fit2[e] = fitness[c];
        fitness[c] = IDV-e;
    }   // end for

    for(e=0; e<IDV; e++){ if(fitness[e]==10)t = e; }

    return t;
}

int tournament(){

    int a, b, c, d, e;
    for(a=0; a<IDV; a++){
        c = random(IDV);
        d = random(IDV);
        if(fitness[c]>fitness[d]) e=c;
        else e=d;
        for(b=0;b<GEN;b++)temparray[a][b] = hovedarray[e][b];
    }
    return 1;
}


int es(){ // utvelgelse istedenfor ruletten.

    int a,b;
    if(ra == 0) rank();

    for(a=0; a<IDV; a++){
        for(b=0; b<GEN; b++){

            if(fitness[a] == 10){
                temparray[0][b] = hovedarray[a][b];
                temparray[1][b] = hovedarray[a][b];
                temparray[2][b] = hovedarray[a][b];
                temparray[3][b] = hovedarray[a][b];
            }
            if(fitness[a] ==9){
                temparray[4][b] = hovedarray[a][b];
                temparray[5][b] = hovedarray[a][b];
                temparray[6][b] = hovedarray[a][b];
            }
            if(fitness[a] ==8){
                temparray[7][b] = hovedarray[a][b];
                temparray[8][b] = hovedarray[a][b];
            }
            if(fitness[a] ==7){
                temparray[9][b] = hovedarray[a][b];
            }
```

```
        }
    }

    return 1;
}


int rulett() // Roulette Selection
{
    int a, b, c, ny, del1=0, del2, tall1, tall2;
    long int sum = 0, tall = 0;

    for(a=0; a<IDV; a++)
    {
        sum = sum + fitness[a];
    } // Accumulates all the fitnesses/100 (max 15 bit, 32767)

    if(sum > 32000){ del1 = sum/2; del2 = sum/2; } // in case it is too big for the random, make 2 partitions

    for(a=0; a<IDV; a++)
    {
        if(del1 != 0)
        { // if total fitness was too big, each partition is treated separately
            tall1 = random(del1);
            tall2 = random(del2);
            tall  = tall1 + tall2;
        }else{  tall = random(sum); } // Finner random tall fra 0 til og med summen av fitnessene.

        // trekker fitnessene fra det random-tallet og velger nytt individ til temparray den fitnessen som "bruker opp" tall.
        ny = 0;

        for(c=0; c<IDV; c++)
        {   // Roulette
            tall = tall - fitness[c];
            if(tall <= 0){ ny=c; c=IDV;} // Checking if the roulette have stopped.
        }   // end for

        for(b=0; b<GEN; b++) temparray[a][b] = hovedarray[ny][b]; // hvis ellitisme vil de to første bli overskrevet senere
    } // end for

    return sum;
}   // end function


int ellitisme(int sjekk)
{
    int nb1=0, nb2=0, fit1=0, fit2=0, fit3=0, a, c, d, e=0, f=1;

    if(ELLI == 0) return 0;

    else if(ELLI == 2)
    {
        if(sjekk == 1) {// ************ Finne de 2 største fitnessverdienes Individnummer.***********
            for(a=0; a<IDV; a++){ fit3 = fitness[a]; // fitness størrelsen blir lagt i fit3 for sjekking

                /* fitvariablene er størrelsen på fitnessene, nb inneholder IDVtallet */
                if(fit3 >= fit1)        // dersom ny fitness er større enn gammel
                {
                    fit2    = fit1; // dytter førsteplass ned på andre
                    fit1    = fit3; // ny fitness inn på førsteplass
                    nb2     = nb1;  // plasseringen blir også gjort om
                    nb1     = a;    // ny plassering inn på førsteplass
                }else if(fit3 > fit2 && fit3 < fit1){
                    fit2    = fit3;
                    nb2     = a;
                }
            }   // end for

            // ***************** Lese inn GEN-strengene inn i elli-arrayet ***********************
            d = nb1; // for å starte med den største
            for(c=0; c<(GEN*2); c++)
            {
                elli[c] = hovedarray[d][e]; e++;
                if(e==GEN && d==nb1){ d = nb2; e = 0; } // end if
            }   // end for
        }
        if(sjekk==2) {//******** Legge fitnessverdiene inn i hovedarray fra elli-array ***********

            for(c=0; c<(GEN*2); c++){
                if(e==GEN){
                    f = 2;
                    e = 0;
                }   // end if

                hovedarray[f][e] = elli[c];
                e++;
            }   // end for
        }   // end if
    } // end if
    return 1;
}   // end function


void kryss()
{
    int     a, b, c, d, j, k, l;
    int     brukt[IDV];

    c = COX/2; // trenger bare halvparten fordi den krysser med den under
```

```
    for(j=0; j<IDV; j++) brukt[j] = 0; // nullstiller brukt-arrayet

    for(k=0; k<c; k++){ //fyller opp brukt arrayet med krysningspunkter / krysser med den under. COX = antall krysninger i en p
        l = random(IDV-1);
        if(brukt[l] == 1) k--;   // På denne måten max et krysningspunkt per Individ!!
        else brukt[l]=1;
    }

    d= IDV-1;
    for(a=0; a<d; a++) // den går bare til den nest siste for å kunne krysse med den under
    {
        if(brukt[a]==1) // Da skal den krysse
        {
            for(b=0; b<GEN; b++)
            {
                if(temparray[a][b] > 255)printf("FEIL før kryss1\n"); // feil fordi int2bin ikke takler det.

                if(b==1)
                {
                    hovedarray[a][b] = temparray[a+1][b];
                    hovedarray[a+1][b] = temparray[a][b]; // hvis feil bytt med n = a+1
                }
                else
                {
                    hovedarray[a][b] = temparray[a][b];   //skal krysses med den under
                    hovedarray[a+1][b] = temparray[a+1][b];  // disse blir nå kun krysset i "skjøtene" mellom gen'ene/posisjone
                }   // end if-else
            }   // end for
        }else {

            for(b=0; b<GEN; b++){
                if(temparray[a][b] > 255) printf("FEIL før kryss2\n");
                hovedarray[a][b] = temparray[a][b];
            }
        }   // end if-else
    }   // end for
}   // end function


void muter()
{
    int a, b, c, d, *p, tall=0;

    for(a=0; a<SMUT; a++)
    {
        b = random(IDV); c = random(GEN);

        if(hovedarray[b][c]>255){printf("**alternativ mutasjon**"); hovedarray[b][c] = random(255);}

        p = int2bin(hovedarray[b][c]);
        d = random(BIT); // muterer nå på pauser og alt.

        if(p[d] == 0) p[d] = 1;
        else p[d] = 0;

        hovedarray[b][c] = bin2int(p);

    }   // end for

}   // end function


int random(int spenn)           // metode som returnerer et tilfeldig tall mellom 0 og parameteren
{
    static  int seed = -1;
            int tall;

    if (seed==-1)               // er denne if-testen nødvendig, ja, pga seed er static?
    {
        seed = GetTickCount();  // for at seed ikke skal bli den samme som forrige, teller den med klokka
        srand(seed);            // henter inn en ny seed velger random av den
    }   // end if

    tall = rand() %spenn;

    return tall;
}


//********************** KONVERTERINGS-FUNKSJONER **************************


int gray2bin(int g)
{
    int flip=0;
    int result=0;

    for (int n=12; n>=0; n--)
    {
        result = result + (flip ^ getBit(g,n)) * pow(2,n);
        flip = getBit(g,n) ^ flip;
    }
    return result;
}   // end gray2bin


int bin2grayGenerell(int b)
{
        // funker for b < 2^29
        int g=0;
```

```
        for (int i=0; i<30; i++)
        {
                g = g + ( ( ((b&(int)pow(2,i))>0) ^ ((b&(int)pow(2,i+1))>0) ) << i );
        }
    return g;
}


int bin2int(int arr[BIT]) // gjør om et binært-array til en int.
{
    int x, tall=0;

    for(x=(BIT-1); x>=0; x--){

        if(arr[x] == 1)

        tall = tall + ((int)pow(2,x));

    }   // end for-loop

    return tall;
}   // end function


int *int2bin(int i) // Det minst signifikante bitet havner på bin[0]
{
    int tall = i, k;
    int sjekk = 4096;

    // nuller ut
    for(k = 0; k < 12; k++)
    {
        bin1[k] = 0;
    }
    if (tall >= sjekk)
    {
        printf("\nTALLET: %d", tall);
        printf("Ditt tall er for stort! Kan ikke oversettes til binaert.\n");
        return NULL;
    }
    else
    {
        for(k=11; k>=0; k--)
        {
            sjekk = sjekk/2;    // halverer sjekk
            if(tall >= sjekk)
            {
                bin1[k] = 1;
                tall = tall - sjekk;
            }// end if
        }   // end for
        return bin1;
    }   // end if-else
}   // end function


/*******************          simulator        *************************/



int runC(double pau, int C1, int C2, int C3, int C4, int Res, int SkrivUt, FILE* filPek)
{
    const  double g         = -9.8;  // m/s*s
    const  double Tinc      = 0.008; // sek
    const  double cyl0kn    = 0.001; // cylinder inkrement

    static double Sy0       = 0.20; // meter
    static double Sx0       = 0.5;
    static double Vy0       = 0.20;
    static double Vx0       = 0;
    static double fot1_X_old;
    static double fot2_X_old;
    static double L1_old;
    static double L2_old;
    static double L3_old;
    static double L4_old;
    static double kropsH_old;
    static int    fot1_iBakken_old;
    static int    fot2_iBakken_old;

    double Sx;
    double Sy; //koordinat til lårfeste
    double Vx; //koordinat til lårfeste
    double Vy;
    double kropsH;
    double fot1_Y;
    double fot2_Y;
    double fot1_X;
    double fot2_X;
    int    fot1_iBakken;
    int    fot2_iBakken;

    if (Res==1)
        {
        L1_old = cyl1_lng(C1,10);
        L2_old = cyl2_lng(C2,10);
        L3_old = cyl3_lng(C3,10);
        L4_old = cyl4_lng(C4,10);

        fotlengder(&fot1_X_old, &fot1_Y, &fot2_X_old, &fot2_Y,  Sx0,Sy0, L1_old,L2_old,L3_old,L4_old, SkrivUt, filPek);
```

```
        if ( fot1_Y < fot2_Y )
            kropsH_old = -fot1_Y;
        else
            kropsH_old = -fot2_Y;

        Sy0     = kropsH_old;
        Sx0     = 0.1;
        Vy0     = 0;
        Vx0     = 0;

        fot1_iBakken = (-fot1_Y) > (Sy0 - 0.00001);
        fot2_iBakken = (-fot2_Y) > (Sy0 - 0.00001);

        return (int)(Sx0*1000);
        }

    for (double t=0; t<pau; t=t+Tinc)
        {
        int lik = likeBevegelser(L1_old,L2_old,L3_old,L4_old, C1,C2,C3,C4);

        double L1 = cyl1_lng(C1,cyl0kn*lik);
        double L2 = cyl2_lng(C2,cyl0kn*lik);
        double L3 = cyl3_lng(C3,cyl0kn*lik);
        double L4 = cyl4_lng(C4,cyl0kn*lik);

        fotlengder(&fot1_X, &fot1_Y, &fot2_X, &fot2_Y,  Sx0,Sy0, L1,L2,L3,L4, SkrivUt, filPek);

        if ( fot1_Y < fot2_Y )
            kropsH = -fot1_Y;
        else
            kropsH = -fot2_Y;

        fot1_iBakken = (-fot1_Y) > (Sy0 - 0.00001);
        fot2_iBakken = (-fot2_Y) > (Sy0 - 0.00001);

        if (fot1_iBakken || fot2_iBakken)
            {           // HVIS I BAKKEN MED NY KROPP
            if (fot1_iBakken)
                {
                if (fot1_X != fot1_X_old)             // HVIS FOT 1 SPARKER BORTOVER
                    Vx0 = -(fot1_X - fot1_X_old)/Tinc;// GI INITSIELL Vx0 fart
                else if ( !(Vy0>0) )                  // HVIS KROPP INITSIELLT IKKE BEVEGER SEG OPPOVER OG
                    Vx0 = 0;                          // FOT IKKE SPARKER BORTOVER :: SETT INITISIELL Vx0 fart til 0
                }
            if (fot2_iBakken)
                {
                if (fot2_X != fot2_X_old)             // HVIS FOT 2 SPARKER BORTOVER
                    Vx0 = -(fot2_X - fot2_X_old)/Tinc;// GI INITSIELL Vx0 fart
                else if ( !(Vy0>0) )                  // HVIS KROPP INITSIELLT IKKE BEVEGER SEG OPPOVER OG
                Vx0 = 0;                              // FOT IKKE SPARKER BORTOVER :: SETT INITISIELL Vx0 fart til 0
                }
             if   (kropsH > kropsH_old)
                 {           // HVIS SPARKER OPPOVER
                 Vy0 = (kropsH - kropsH_old)/Tinc;     // GI INITSIELL Vy0 fart
                 }
             }

        //---------------YYYYYYYYYYYYYYYYYYY---------------------------
        Sy = Tinc*Tinc*g/2 + Vy0*Tinc + Sy0;              // FINN NY Y POSISJON
        Vy = Tinc*g + Vy0;                                // FINN NY Y FART

        if ( (Sy + 0.00001) < kropsH )                    // HVIS KLEMT NED I BAKKEN
            Sy = kropsH;                                  // REIS OPP
        //---------------XXXXXXXXXXXXXXXXXXX---------------------------
        Sx =  Vx0*Tinc + Sx0;                             // FINN NY X POSISJON
        Vx =  Vx0;                                        // FINN NY X FART
        //-------------------------------------------------------------

        fot1_X_old = fot1_X;
        fot2_X_old = fot2_X;
        fot1_iBakken_old = fot1_iBakken;
        fot2_iBakken_old = fot2_iBakken;
        L1_old = L1;
        L2_old = L2;
        L3_old = L3;
        L4_old = L4;
        kropsH_old = kropsH;

        Sy0 = Sy;
        Sx0 = Sx;
        Vy0 = Vy;
        Vx0 = Vx;
        }
        return (int)(Sx0*1000);
}


//********************************************

inline int likeBevegelser(double L1_old,double L2_old,double L3_old,double L4_old,int C1,int C2,int C3,int C4)
    {
    int lik;

    if ( (L1_old == L3_old) && (L2_old == L4_old) && (C1==C3) && (C2==C4) )
        lik = 2;
    else
        lik = 1;

    return lik;
    }
```

```
//********************************************

inline int likeBevegelser(double L1_old,double L2_old,double L3_old,double L4_old,int C1,int C2,int C3,int C4)
    {
    int lik;

    if ( (L1_old == L3_old) && (L2_old == L4_old) && (C1==C3) && (C2==C4) )
        lik = 2;
    else
        lik = 1;

    return lik;
    }

//********************************************
//********************************************

inline void fotlengder(double* X1, double* Y1, double* X2, double* Y2, double Sx, double Sy, double L1, double L2, double L3, do
    {
    // L1 - lengde på lårsylA
    // L2 - lengde på leggsylA
    // L3 - lengde på lårsylB
    // L4 - lengde på leggsylB
    // X1, Y1 hæl relativ til lårfeste fot A (origo)
    // X2, Y2 hæl relativ til lårfeste fot B (samme origo)
    const double TVE_lng = 0.03;
    const double TVE_BAK_lng = 0.01;

    const double LAA_lng = 0.081;
    const double LAA_lngN = 0.01;

    const double LEGG_lng = 0.09;
    const double LEGG_lngN = 0.015;
    //-------------------------------

    double VinA = acos((TVE_lng*TVE_lng + L1*L1 - LAA_lng*LAA_lng) / (2*L1 * TVE_lng));

    double CyHoy1 =  L1 *sin( VinA );

    double CyBakover1 = L1 *cos( VinA );

    double VinC = atan(CyHoy1/(CyBakover1-TVE_lng));

    double LarBak = cos(VinC)*(LAA_lng+LAA_lngN);
    double LarHoy = sin(VinC)*(LAA_lng+LAA_lngN);

    double LarX = - LarBak;
    double LarY = - LarHoy;

    double m = sqrt(LarHoy*LarHoy + (LarBak-TVE_BAK_lng)*(LarBak-TVE_BAK_lng));

    double VinE = acos((m*m + L2*L2 - LEGG_lngN*LEGG_lngN) / (2*m * L2));

    double VinF = acos(LarHoy/m);

    double VinD = PI - VinF - VinE - PI/2;

    double CyBx2 = - TVE_BAK_lng - cos(VinD)*L2;
    double CyBy2 = - sin(VinD)*L2;

    VinC = asin((LarX-CyBx2)/LEGG_lngN);

    *X1 = LarX + sin(VinC)*LEGG_lng;
    *Y1 = LarY - cos(VinC)*LEGG_lng;

    if (SkrivUt>0)
        {
        fprintf(filPek, "%d %d %d %d\n", (int)((LarX+Sx)*1000), (int)(Sx*1000), (int)((LarY+Sy)*1000), (int)(Sy*1000) ); // LÅR
        fprintf(filPek, "%d %d %d %d\n", (int)((CyBx2+Sx)*1000), (int)((*X1+Sx)*1000), (int)((CyBy2+Sy)*1000), (int)((*Y1+Sy)*10

        if (SkrivUt==2)
            {
            fprintf(filPek, "%d %d %d %d\n", (int)((Sx-TVE_BAK_lng)*1000), (int)((Sx+TVE_lng)*1000), (int)(Sy*1000), (int)(Sy*10
            fprintf(filPek, "%d %d %d %d\n", (int)((Sx+TVE_lng-CyBakover1)*1000), (int)((Sx+TVE_lng)*1000), (int)((Sy-CyHoy1)*10
            fprintf(filPek, "%d %d %d %d\n", (int)((CyBx2+Sx)*1000), (int)((Sx-TVE_BAK_lng)*1000), (int)((CyBy2+Sy)*1000), (int)
            }
        }

    //------------------------------------------

    VinA = acos((TVE_lng*TVE_lng + L3*L3 - LAA_lng*LAA_lng) / (2*L3 * TVE_lng));

    CyHoy1 =  L3 *sin( VinA );

    CyBakover1 = L3 *cos( VinA );

    VinC = atan(CyHoy1/(CyBakover1-TVE_lng));

    LarBak = cos(VinC)*(LAA_lng+LAA_lngN);
    LarHoy = sin(VinC)*(LAA_lng+LAA_lngN);

    LarX = - LarBak;
    LarY = - LarHoy;

    m = sqrt(LarHoy*LarHoy + (LarBak-TVE_BAK_lng)*(LarBak-TVE_BAK_lng));

    VinE = acos((m*m + L4*L4 - LEGG_lngN*LEGG_lngN) / (2*m * L4));

    VinF = acos(LarHoy/m);

    VinD = PI - VinF - VinE - PI/2;
```

```
    CyBx2 = - TVE_BAK_lng - cos(VinD)*L4;
    CyBy2 = - sin(VinD)*L4;

    VinC = asin((LarX-CyBx2)/LEGG_lngN);

    *X2 = LarX + sin(VinC)*LEGG_lng;
    *Y2 = LarY - cos(VinC)*LEGG_lng;

    if (SkrivUt>0)
        {
        fprintf(filPek, "%d %d %d %d\n", (int)((LarX+Sx)*1000), (int)(Sx*1000), (int)((LarY+Sy)*1000), (int)(Sy*1000) ); // LÅR
        fprintf(filPek, "%d %d %d %d\n", (int)((CyBx2+Sx)*1000), (int)((*X2+Sx)*1000), (int)((CyBy2+Sy)*1000), (int)((*Y2+Sy)*10

        if (SkrivUt==2)
            {
            fprintf(filPek, "%d %d %d %d\n", (int)((Sx+TVE_lng-CyBakover1)*1000), (int)((Sx+TVE_lng)*1000), (int)((Sy-CyHoy1)*10
            fprintf(filPek, "%d %d %d %d\n", (int)((CyBx2+Sx)*1000), (int)((Sx-TVE_BAK_lng)*1000), (int)((CyBy2+Sy)*1000), (int)
            }
        }
}


inline double cyl1_lng(int C, double incr)
    {
    const double CYL_lng_max = 0.103;
    const double CYL_lng_min = 0.091;

    static double L = CYL_lng_min;

//------------------------------


    if (C==1)
        {
        L = L + incr;
        if (L > CYL_lng_max)
            L = CYL_lng_max;

        return L;
        }

    if (C==0)
        {
        L = L - incr;
        if (L < CYL_lng_min)
            L = CYL_lng_min;

        return L;
        }

    return 0;
    }


inline double cyl2_lng(int C, double incr)
    {
    const double CYL_lng_max = 0.082;
    const double CYL_lng_min = 0.076;

    static double L = CYL_lng_min;

    //------------------------------

    if (C==1)
        {
        L = L + incr;
        if (L > CYL_lng_max)
            L = CYL_lng_max;

        return L;
        }

    if (C==0)
        {
        L = L - incr;
        if (L < CYL_lng_min)
            L = CYL_lng_min;

        return L;
        }

    return 0;
    }


inline double cyl3_lng(int C, double incr)
    {
    const double CYL_lng_max = 0.103;
    const double CYL_lng_min = 0.091;

    static double L = CYL_lng_min;

    //------------------------------

    if (C==1)
        {
        L = L + incr;
        if (L > CYL_lng_max)
            L = CYL_lng_max;
```

```
        return L;
        }

    if (C==0)
        {
        L = L - incr;
        if (L < CYL_lng_min)
            L = CYL_lng_min;

        return L;
        }

    return 0;
    }


inline double cyl4_lng(int C, double incr)
    {
    const double CYL_lng_max = 0.082;
    const double CYL_lng_min = 0.076;

    static double L = CYL_lng_min;

    //-----------------------------

    if (C==1)
        {
        L = L + incr;
        if (L > CYL_lng_max)
            L = CYL_lng_max;

        return L;
        }

    if (C==0)
        {
        L = L - incr;
        if (L < CYL_lng_min)
            L = CYL_lng_min;

        return L;
        }

    return 0;
    }
/*****************************************************************************************/
```

Bibliography

[1] Gregory Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto and M. Fujita, *"Evolving robust gaits with AIBO"*, IEEE International Conference on Robotics and Automation, San Fransisco USA, 2000

[2] David Goldberg, *"Genetic Algorithms in search, optimization and machine learning"*, Addison Wesley Press Book. pp.2-3, 1989

[3] Tom Michael Mitchell, *"Machine Learning"*, McGraw-Hill Book Company. pp.81-124, 1997

[4] Stefano Nolfi and Dario Floreano, *"Evolutionary Robotics"*, Book in Biology, Intelligence, and Technology of self-organizing Machines, MIT Press Massachusetts, 2000

[5] E. Cantu-Paz, *"A survey of parallel genetic algorithms"*, Calculateurs Parallels, Paris France, 1998

[6] Charles Darwin after John Murray, *"On the origin of species by means of natural selection, or preservation of favoured races in the struggel for life"*, 1859

[7] William A. Dembski, *"The Bridge Between Science and Theology"*, Inter Varisty Press, 1999

[8] Mans Ullerstam and Makoto Mizukawa, *"Teaching robots behavior patterns by using reinforcement learning"*, SICE Annual Conference, Sapporo Japan, 2004

[9] Richard S. Sutton and Andrew G. Barto, *"Reinforcement learning: An introduction"*, A Bradford Book. MIT Press, 1998

[10] T.D. Barfoot, E.J.P. Earon and G.M.T. DÉleuterio, *"A step in the right direction. Learning Hexapod Gaits Through Reinforcement"*, International Symposium on Robotics, Montreal Canada, 2000

[11] C.J.C.H. Watkins, *"Learning from Delayed Rewards"*, PhD Thesis, Cambridge University, Cambridge England, 1989

[12] M. Anthony Lewis, Andrew H. Fagg and George A. Bekey, *"Genetic Algorithms for Gait Synthesis in a Hexapod Robot"*, Recent Trends in Mobile Robots. pp.317 - 331, World Scientific, New Jersey USA, 1994

[13] MIT Leg Lab,
http://www.ai.mit.edu/projects/leglab/home.html (May 2006)
http://www.ai.mit.edu/projects/leglab/publications/publications-main.html (May 2006)

[14] Boston Dynamics,
http://www.bdi.com/content/sec.php?section=robotics (May 2006)

[15] Hisashi Fukuda and Tomiyuki Arakawa, *"Natural motion trajectory generation of biped locomotion robot using genetic algorithm through energy optimization"*, Proceedings of IEEE International Conference on Systems, Man and Cybernetics. pp.1495, Beijing China, 1996

[16] Jim Tørresen, *"A divide-and-conquer approach to evolvable hardware"*, Proceedings of the ICES, volume 1475 of Lecture Notes in Computer Science. Springer-Verlag. pp.57-65, Lausanne Switzerland, 2000

[17] P.C. Haddow and G. Tufte, *"An evolvable hardware FPGA for adaptive hardware"*, Proceedings of Congress on Evolutionary Computation, San Diego USA, 2000.

[18] Tatiana Kalganova, *"Bidirectional Incremental Evolution in Evolvable Hardware"*, Proceedings of The NASA/DoD Workshop on Evolvable Hardware, California USA, IEEE Computer Society, 2000.

[19] Tetsuya Higuchi, Masaya Iwata, Didier Keymeulen, Hidenori Sakanashi, Masahiro Murakawa, Isamu Kajitani, Eiichi Takahashi, Kenji Toda, Mehrad Salami, Nobuki Kajihara, and Nobuyuki Otsu, *"Real-world applications of analog and digital evolvable hardware"*, IEEE Transactions on Evolutionary Computation. Vol.3. pp.220 - 235, 1999

[20] Xin Yao and Tetsuya Higuchi, *"Promises and Challenges of Evolvable Hardware"*, Transactions on Systems, Man, and Cybernetics, part C. vol.29. no.1, 1999

[21] Jim Tørresen, *"Two-Step Incremental Evolution of a Prosthetic Hand Controller Based on Digital Logic Gates"*, International Conference on Evolvable Hardware, Tokyo Japan, 2001

[22] Julian Miller, Dominic Job and Vesselin K. Vassilev, *"Principles in the Evolutionary Design of Digital Circuits - Part 1"*, Genetic Programming and Evolvable Machines, 1999

[23] T.Back and F.Hoffmeister and H.Schewefel, *"A Survey of Evolution Strategies"*, University of Dortmund, Germany, 1991

[24] John R. Koza, Forrest H. Bennett, David Andre and Martin A. Keane, *"Automatic Design of Analog Electrical Circuits using Genetic Programming"*, Book in Intelligent Data Analysis in Science. Oxford University Press. pp.172–200, 2000

[25] Tom M. Mitchell, *"Machine Learning"*, Book in Machine Learning, McGraw-Hill Book Company. pp.262-266, 1997

[26] Mats Høvin,
http://heim.ifi.uio.no/ matsh/591188/index.html

[27] David Goldberg, *"Genetic Algorithms in search, optimization and machine learning"*, Addison Wesley Book. pp.100-101, 1989

[28] David Goldberg, *"Genetic Algorithms in search, optimization and machine learning"*, Addison Wesley Book. pp.59-75, 1989

[29] John Holland, *"Adaptation in Natural and Artificial Systems"* University of Michigan, 1992 (first published in 1975)

[30] Yuval Davidor, *"A Naturally Occurring Niche and Species Phenomenon, The Model and First Results."* In proceedings of the International Conference Genetic Algorithms, San Diego USA, 1991

[31] http://en.wikipedia.org/wiki/Genetic_algorithms. (May 2006)

[32] J.D. Lohn, D.S. Linden, G. S. Hornby, W. F. Kraus, A. Rodriguez and S. Seufert, *"Evolutionary Design of an X-Band Antenna for NASA's Space Technology 5 Mission"*, In proceedings of IEEE Antenna and Propagation Society International Symposium and USNC/URSI National Radio Science Meeting, Vol.3 pp.2313-2316, 2004

[33] B. Sharon, G. Beals, *"Software au naturel."*, Newsweek pp.70, 1995

[34] Jim Tørresen, *"An evolvable hardware tutorial"*, Proceedings of the International Conference on Field Programmable Logic and Applications, Antwerp Belgium, 2004

[35] David Goldberg, *"Genetic Algorithms in search, optimization and machine learning"*, Addison Wesley Book. pp.10-12, 1989

[36] David Goldberg, *"Genetic Algorithms in search, optimization and machine learning"*, Addison Wesley Book. pp.121-122, 1989

[37] David Goldberg, *"Genetic Algorithms in search, optimization and machine learning"*, Addison Wesley Book. pp.115-116, 1989

[38] Kenneth A. De Jong and Mitchell A. Potter, *"Evolving Complex Structures via Cooperative Coevolution"*, Proceedings of the Fourth Annual Conference on Evolutionary Programming, MIT Press. pp.307-317, 1995

[39] Dario Floreano and Francesco Mondada, *"Hardware solutions for evolutionary robotics"*, Proceedings of the First European Workshop on Evolutionary Robotics. Springer-Verlag, Berlin Germany, 1998

[40] Sadao Fukunaga and Andrew B. Kahng, *"Improving the performance of evolutionary optimization by dynamically scaling the evolution function"*, Proceedings of the IEEE Conference on Evolutionary Computation, IEEE Press. Vol.1 pp.182-187, Perth Australia, 1995.

[41] Jim Tørresen, *"Evolving both Hardware Subsystems and the Selection of Variants of such into an assembled System"*, Proceedings of the European Simulation Multiconference, pp.451-457, Darmstadt Germany, 2002

[42] Mitchell A. Potter and Kenneth A. De Jong, *"Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents"*, Evolutionary Computation, MIT Press, 2000

[43] Gary B. Parker, *"Cyclic Genetic Algorithm with Conditional Branching in a Predator-Prey Scenario"*, Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, Waikoloa Hawaii, 2005

[44] Jim Tørresen, *"Exploring Knowledge Schemes for Efficient Evolution of Hardware"*, Proceedings of NASA/DoD Conference on Evolvable Hardware, Seattle USA, 2004

[45] Jim Tørresen, *"A Scalable Approach to Evolvable Hardware"*, Genetic Programming and Evolvable Machines, Vol.3 No.3 pp.259-282, 2002

[46] Jim Tørresen, *"Evolving Multiplier Circuits by Training Set and Training Vector Partitioning"*, Proceedings of the International Conference on Evolvable Hardware, Springer LNCS 2606. pp.228-237, Trondheim Norway, 2003

[47] Jim Tørresen, *"Possibilities and Limitations of Applying Evolvable Hardware to Real-World Applications"*, International Conference on Field Programmable Logic and Applications, Villach Austria, 2000

[48] Moritoshi Yasunaga, Takahiro Tsuzuku, Kentaro Ushiyama, Ikuo Yoshihara and Jung H. Kim, *"Evolvable Reasoning Hardware: It's Application to the Genome Informatics"*, Proceedings of Congress on Evolutionary Computation, Seoul Korea, 2001

[49] Moritoshi Yasunaga, Jung H. Kim and Ikuo Yoshihara, *"Evolvable Reasoning Hardware: It's Prototyping and Performance Evaluation"*, Proceedings of Genetic Programming and Evolvable Machines, pp.211-230, Seoul Korea, 2001

[50] Moritoshi Yasunaga, Kentaro Ushiyama, Noriyuki Aibe, Hidetoshi Fujiwara, Ikuo Yoshiyara and Jung H. Kim, *"An Evolutionary Kernel-Based Reasoning System Using Reconfigurable VLSI: It's Hardware Prototyping and Application to the Aplicing Boundary Problem"*, Proceedings of IEEE Congress on Evolutionary Computation, Honolulu Hawaii, 2002

[51] Morse Sipper, Eduardo Sanchez, Daniel Mange, Marco Tomassini, Andrès Pèrez-Uribe and Andrè Stauffer, *"A Polygenetic, Ontogenetic and Epigenetic View of Bio-Inspired Hardware Systems"*, IEEE Transaction Evolvable Computation. Vol.1, 1997

[52] Paul J. Darwen and Xin Yao, *"Speciation as Automatic Categorical Modularization"*, IEEE Transactions on Evolutionary Computation, Vol.1 pp.101-108, 1997