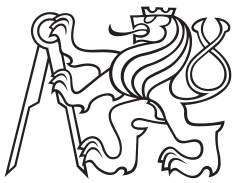
Bachelor's thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Virtual Creatures simulated in Framsticks

Vít Růžička

Program: Open Informatics
Field: Computer and Information Science

Spring 2014
Supervisor: Ing. Zdeněk Buk, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Vít R ů ž i č k a

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Virtual Creatures Simulated in Framsticks

Guidelines:

Get in touch with the experiment of Karl Sims "Evolving Virtual Creatures" a repeat this work with help of simulation capabilities of Framstics. After discussion with thesis supervisor choose appropriate evolutionary algorithm and experiment with mutation and recombination steps of evolution. Suggest suitable representation of a genome and discus the possibilities of using hybrid evolutionary algorithm.

Bibliography/Sources:

[1] Karl Sims - "Evolving Virtual Creatures" (1994) [http://karlsims.com/papers/siggraph94.pdf]
[2] Karl Sims - "Evolving 3D Morphology and Behavior by Competition" (1994) [http://karlsims.com/papers/alife94.pdf]
[3] Nicolas Lassabe - "A New Step for Artificial Creatures" (2007) [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.149.4345&rep=rep1&type=pdf]
[4] Dan Lessin - "Open-Ended Behavioral Complexity for Evolved Virtual Creatures" (2013) [http://nn.cs.utexas.edu/downloads/papers/lessin.gecco13.pdf]

Bachelor Project Supervisor: Ing. Zdeněk Buk, Ph.D.

Valid until: the end of the summer semester of academic year 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Vít R ů ž i č k a

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Virtuální bytosti simulované ve Framsticks

Pokyny pro vypracování:

Seznamte se s pokusem Karla Simse "Evolving Virtual Creatures" a zopakujte tento pokus za využití simulačního prostředí Framsticks. Po konzultaci s vedoucím práce zvolte vhodný evoluční algoritmus a zaměřte své experimenty na modifikace mutačních a rekombinačních operátorů. Navrhněte vhodnou reprezentaci genomu a diskutujte potenciální využití hybridního evolučního algoritmu.

Seznam odborné literatury:

[1] Karl Sims - "Evolving Virtual Creatures" (1994) [http://karlsims.com/papers/siggraph94.pdf]
[2] Karl Sims - "Evolving 3D Morphology and Behavior by Competition" (1994) [http://karlsims.com/papers/alife94.pdf]
[3] Nicolas Lassabe - "A New Step for Artificial Creatures" (2007) [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.149.4345&rep=rep1&type=pdf]
[4] Dan Lessin - "Open-Ended Behavioral Complexity for Evolved Virtual Creatures" (2013) [http://nn.cs.utexas.edu/downloads/papers/lessin.gecco13.pdf]

Vedoucí bakalářské práce: Ing. Zdeněk Buk, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2014

Acknowledgement / Declaration

I would like to thank my supervisor Ing. Zdeněk Buk, Ph.D. for assistance and gentle shoving in the right direction when I needed it.

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

.....
Vít Růžička

Prague, 21 May 2014

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

.....
Vít Růžička

V Praze, 21. května 2014

Abstrakt / Abstract

Tato bakalářská práce se zaměřuje na aplikaci evolučního algoritmu na dobře známý problém vývoje virtuálních bytostí. Rozebírá vhodnost použití hybridního algoritmu s diferenciální evolucí pro obdobnou úlohu. V práci je popsána struktura vytvořené abstraktní vrstvy nad genotypem f0 volně přístupného programu „Framsticks“, která umožňuje snazší ovládání průběhu evoluce. Práce zkoumá vhodné hodnoty parametrů v několika ukázkových experimentech s různým nastavením fitness funkce. Závěr zhodnocuje podobu a chování vzniknuvších jedinců.

Klíčová slova: Virtuální bytosti; Framstics; evoluční algoritmy; diferenciální evoluce.

Překlad titulu: Virtuální bytosti simulované ve Framsticks

This bachelors thesis focuses on application of evolutionary algorithm to the well-known problem of virtual creatures evolution. It analyzes the capabilities of hybrid algorithm with differential evolution on this task. This thesis describes the structure of implemented layer of abstraction above the f0 genotype of free program „Framsticks“, which is designed to help control the progress of evolution. It explores the optimal values of parameters in several exemplary experiments with various settings of fitness function. The conclusion discusses the appearance and behavior of resulting individuals.

Keywords: Virtual creatures; Framstics; evolutionary algorithms; differential evolution.

Contents /

Tables / Figures

Chapter 1
Introduction

1.1 Structure of the thesis

In order to guarantee capable evolutionary algorithm we need to study both the theoretical and the implementationary side of things. We take a look at couple of interesting studies in the area of virtual creatures in chapter 1 after this introduction. At first we mention the initial work of Karl Sims and the consequential papers that expanded upon his work. We mention the theoretical background of evolutionary algorithms in chapter 2. We study the possibilities and usage of differential evolution. In chapter 3 we talk about the implementation of said algorithms and about the abstraction layer made on top of Framsticks genotype. We show exemplary experiments in chapter 4 and discuss the results and sample individuals. All is summarized in chapter 5 and possible future studies are suggestes.

1.2 Karl's Sims Virtual Creatures – How it all started

Karl Sims wrote his paper *Evolving Virtual Creatures* [1] in 1994 and since then many other researches were inspired by his work in this field. He used the concept of creatures living in virtual spaces. These creatures consisted both from body and brain. Physical parts (boxes of various proportions) and neural system that controlled those physical components.

He used the terms *genotype* and *fenotype*, where genotype represents code by which are creatures completely described (sort of like human DNA, the source code for construction of human being – with the exception, that virtual creatures created by code are always the same). Phenotype is then what gets constructed – individual built from the blueprints of genotype.

Karl Sims then used by nature inspired evolutionary algorithm to develop sophisticated creatures from the absolute chaos of random generated initial population. He used mutation and crossover of virtual creature genotypes to develop more complex specimen. How exactly he did it, what representation he used for virtual creatures and how he defined virtual world rules and fitness evaluation function can be read in his paper [1].

In his second paper *Evolving 3D Morphology and Behavior by Competition* [2], he adressed the idea of evolution through competition. In nature we can see this principle as species are evolving in the way best suitable for survival in their habitat. This evolution is usually very gradual and slow. Rate of evolution is majorly boosted when the population is faced with some dangerous outside force. Then the very existence of entire species is in danger and individuals are forced to change and adapt in order to survive in this new environment. One can use this principle to make specimen of various species compete each other. We expect to see evolution of strategies and counter strategies in cycles of co-evolution.

There are many possible ways to approach this competition. Very important aspect is the coupling of pairs from the entire population to compete each other. We may use competition evaluation between creatures of only one class or we can have two or more competing species that battle between each other while evolving amidst individuals of the same species. Other problem we have to keep in mind is the possible time consumption for each round of battles to be evaluated. For example having each creature combat every possible opponent would be very time demanding. We can use randomized pairs but that may yield random results. Perhaps the best approach is to have competition of all creatures of one class to fight only the best specimen of other class.

His work inspired many other researches which used his altered concept. Many tried following his visions and thoughts of future development of virtual creatures. He suggested trying different fitness functions, evolution through competition and he also mentioned the possibility of constraining creatures by the rule that they must be constructible as real robots.

1.3 The story so far

1.3.1 Hardware creatures

Those limited by real world constrains but probably closest to practical usage. The more specialized case of this approach would be some sort of autonomous workstation that would adapt to its surroundings and perform various tasks by assembling real world virtual creature robots from modular parts.

The best example of this principle would be the work *Automatic design and Manufacture of Robotic Lifeforms* [3] which focused on evolution of virtual creatures that were automatically manufactured by rapid prototyping machine. They willingly minimised the influence of human operators in order to have their system as autonomous as possible. The only human input was the starting definition of simulated world (as close to the reality as was computationally possible) and the assembling of printed robot parts with motors. Their virtual creatures and then manufactured robots consisted from bars, joints and linear actuators (they didn't include any sensor parts). Their reason behind autonomous behaviour of the entire system was that the most difficult task is the planning and manifacture of robots. Apart from mass production it isn't cost efficient to design each robot and it would be useful to have them develop themselfs in virtual space.

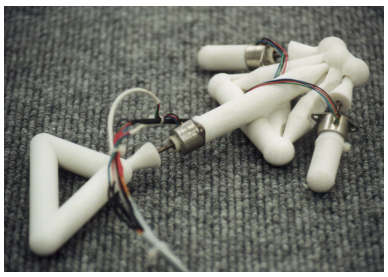


Figure 1.1. Example of virtual creature constructed in real world. Image used from webpage of project *Golem*[1])described in [3].

[1]) `http://www.demo.cs.brandeis.edu/golem/`

1.3.2 Evolution limited to neurons and musles

In multiple papers the evolution is limited to certain part of virtual creature. We can use bodies predesigned by human designer and bring them alive through the evolution of neural network. In the work *Evolving Physically Simulated Flying Creatures for Efficient Cruising* [4] we can see the task to learn how to fly in stable manner in an avian-like bodies by evolution of wing shape and their controllers while wing root and testing environment remains the same.

In the paper *Flexible Muscle-Based Locomotion for Bipedal Creatures* [5] we can see the task of adapting a natural-looking motion of general bipedal bodies. This could be used to animate human-like figure or even fictional body types into motion, which is shown on the example of dinosaur creatures. For this evolutionary task there is no need for solid data gain from motion capture, which would not be possible in the case of dinosaurs. Furthermore this task was expanded by variating the fitness function and world rules for several variants of generated movement. The parameter of required target speed achieved the result of simply walking or running creatures. In other experiment creatures had to follow path and learn how to steer properly or move on uneven terrain. The last test was for creatures to adapt stable enough movement to endure various external disturbances (in the form of randomly genereted boxes being tossed at moving creature).

1.3.3 Different aproaches to task definition

1.3.4 Stairs and pylons

In the paper *A New Step for Artificial Creatures* by Nicolas Lassabe [6] we can see various experiments with the usage of evolved virtual creatures. We can see more complex worlds having obstacles for creatures to pass through. Starting with the usual walking creatures we see the experiment to generate creatures walking in one specific direction. These creatures usually manifest far more symmetricity than those not constrained by the rule of one direction movement. Other task was to climb stairs with various step height and traverse across pylons with changing space between each other (where falling from pylons is frowned upon and punished severely by dividing fitness by ten). Aspects of these experiments are joined in one that forces virtual creatures to live and walk in completely randomly generated terrain.

1.3.5 Skateboarding creatures

Very interesting is the experiment with skateboarding creatures from the same paper [6]. These have to control predesigned block with the behaviour similar to skateboard (movable box with wheels). Creatures have to develop the capability of riding and pushing this foreign object without letting it go. Interesting here is the interaction with unchanging tool – the evolutionary learning process to use tools correctly.

1.3.6 Behavioral approach

Its often useful to think about how the programmer of these evolving systems can influence (by chance or by willful attempt) the result of evolutionary algorithms. The obvious bottleneck would be the evalutation fitness function, ways how the world around creatures functions or even the computational limitations brought by chosen representation of creature's genotype. This control of virtual population can be reduced in

order to find the most general solution to our solved problem – leaving doors opened for designs that we would not think about ourselves.

On the other hand, this control can be admitted and used to create creatures of more complex behaviour that suits our needs. In the recent work of Dan Lessin [7] named *Open-Ended Behavioral Complexity for Evolved Virtual Creatures* we can see the effort to construct some high-level program for creatures to evolve into. There we have the *syllabus* that contains small broken-down tasks of the complex behaviour and the order in which creatures learn these small fragments. Once creature has mastered one individual task sufficiently enough then comes the *encapsulation* process in which we lockdown created body and neuron parts. Then we go to next task and we make creatures evolve new neural connections and parts that will fulfill this next task. We cannot break-down already *encapsulated* segments of our creature. This leads to approach of human to control the very abstract task creation of *syllabus* that leads to emergence of behavioral encapsulated functions this creature is capable of. By the brute force of evolution creatures learn how to complete this abstract program.

Chapter 2
Evolutionary algorithms

So from the first chapter we know about virtual creatures, what they are and how can they be developed with the help of evolutionary algorithms. In this chapter, we will be using the terminology and general idea of evolutionary algorithms from sources like [8–9].

2.1 Evolutionary algorithms

Evolutionary algorithms are often used in optimalization tasks and we can look at the task of finding the best specimen, virtual creature, as search in world of many dimensions (corresponding with the number of parameters of creature's genotype) with one maximized value – fitness. It can be said, that we are maximizing function that is a simulator with input of genotype and one output of fitness.

Given the fact, that individual can be represented by very high number of parameters, the space that this fitness function represents can be very irregular or even pathological. There are no guaranties of its continuity, there can be great quantities of local extremes and there even might not be one global extreme and solution. Imagine a situation when changing one parameter to greater value will always lead to better fitness – for example while building the tallest creature. If simulator permits tall creatures to exist (and doesn't for example force them to collapse under its weight), the creature will grow infinitely.

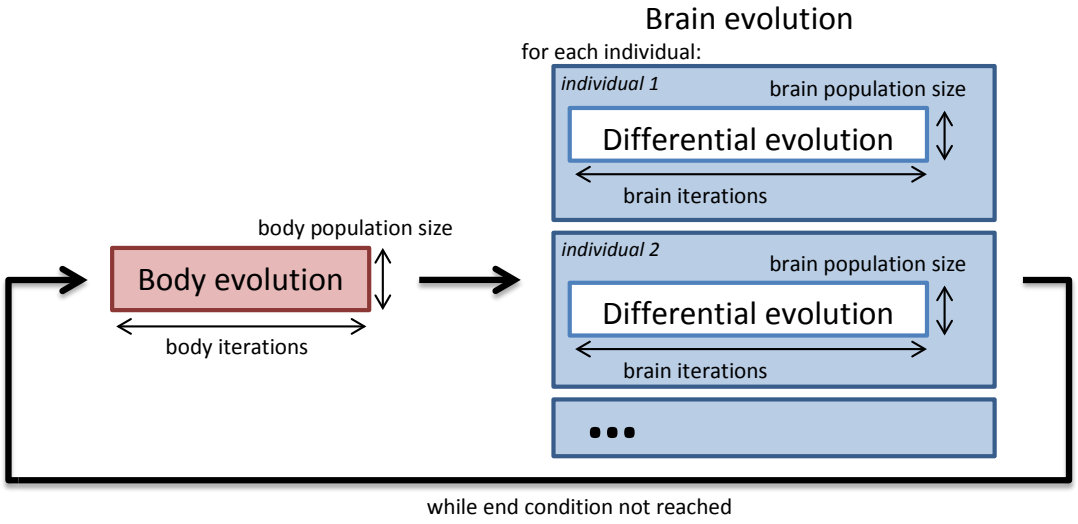


Figure 2.1. Scheme of separated body and brain evolution. Each phase has its own set of parameters – here we can see the most basic ones, number of iterations and population size.

And for these kinds of tasks are evolutionary algorithms suitable. There is of course vast majority of different approaches to evolutionary algorithms and we must choose carefully, which one is the best for our case. For this bachelor's thesis the task is to implement hybrid evolutionary algorithm.

2.1.1 Hybrid algorithms

Hybrid algorithms combine mutation and crossover process with local search methods. Evolutionary algorithm can be seen as implementation of natural model of genetic evolution. The actual life of each individual, learning how to live can be understood as finding local optimum. This idea is implemented for example in *Memetic algorithm* which uses local search to improve each crossovered or mutated individual. We use the same idea in separating body and brain evolution. Brain evolution serves us as tuning phase, where we use *Differential evolution* 2.3. The general idea is, that for each individual from current population, we start new evolutionary algorithm in which we have multiple configurations of brain (matrix of weights) for population. See schematic representation of this idea in figure 2.1 (for more details see [8, 10]).

2.2 Generic evolutionary algorithm

In evolutionary algorithms we often follow the process inspired by nature's evolution and its simple form can be seen on attached diagram 2.2.

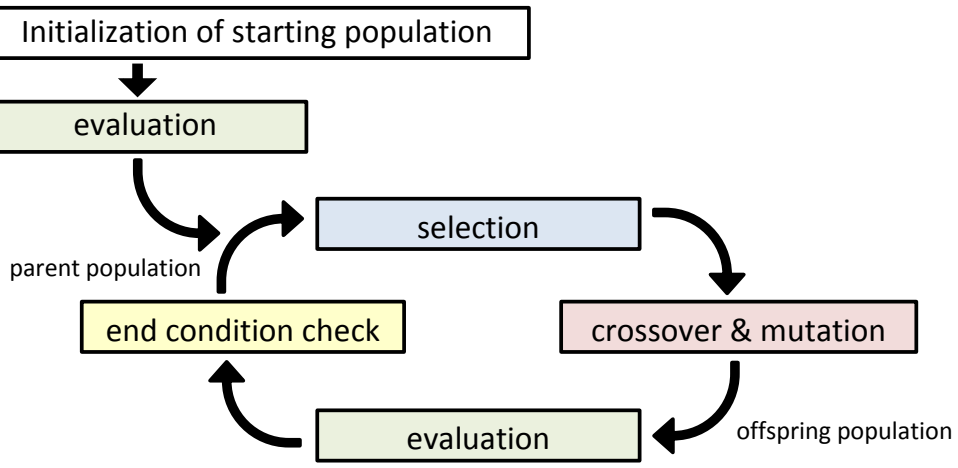


Figure 2.2. General evolutionary algorithm diagram (structure which is commonly used in literature [8–9, 11]).

We start with couple of randomly generated (or carefully designed) individuals in what we call a starting population. We evaluate it (assign fitness to each individual) and then we come into the algorithm's main cycle.

Between states which virtual creature comes through (and we can think of these like specific configurations of parameters) there are what we call genetic operators. These often are mutation or some kind of crossover process. Mutation corresponds to asexual reproduction in nature, when the individual is modified by mutation of one or more of its parameters. Crossover on the other hand takes group of parents (usually a couple) and creates from them new offspring. All these new individuals are now part of the new population.

This is the part of algorithm that differentiates the most with currently solved task. Here we can use heuristics, here we are bound by limitations and here we can choose different approaches. Usually we use the process called **elitism** which means, that we always take certain number of best individuals from parent population and force their way into the next population unchanged. The more sophisticated version of this could be implementing aging process and allow survival of only parents younger (in "iterations" years) that certain age. In the process of determining which individuals in parent population will be chosen for mutation and which couples will be used as parents, we use methods of selection and sampling.

2.2.1 Alterations on generic evolutionary algorithm

While working with virtual creatures and standalone evaluator Framsticks (for more details see next chapter 3) its more convenient for us to evaluate whole populations of creatures than to evaluate every single one separately (the reason here is the number of communications with Framsticks client), however they are still simulated in separated instances (so members of population cannot influence themselves amongst population). In this case we can call genetic operator the whole transition between parent population into next iteration's population.

To ensure having non decreasing fitness over the course of iterations, we implemented elitism of minimal one individual from parent population. For greater diversity and to avoid dominance of one creature's clones (the one with the best fitness) we implemented fitness function transformation into **ranking**. Instead of depending on fitness alone, which could lead to unwanted preference of one individual with fitness much higher than the rest, we sort them by their fitness and associate new ranking by their order. Then we count a new fitness in proportion to its ranking. With N creatures in population we would use this formula to assign transformed fitness to i-th creature (with $i \in 1 \rightarrow N$):

$$fitness_i = \frac{order_i}{\sum_{k=1}^{N} k} \tag{1}$$

With the transformation of fitness function we used the roulette selection as **sampling** method. This method places all fitness (or all transformed values) into roulette space of 0–1 in ratio to their transformed fitness.

$$fitness_i = \frac{fitness_i}{\sum_{i=1}^{N} fitness_i} \tag{2}$$

Instead of selecting for example the first N best ones, this gives chance even to the seemingly bad individuals. For more detailed analysis of selection methods see relevant chapters in [10]. That leaves us open for solutions other than the current best one is following and helps the diversity of evolution. **Roulette selection** in hand with fitness transformation by ranking should be enough. Of course the discipline of evolutionary algorithms often gives us many possible approaches and this chosen combination could be tested with other possible implementations (see future work chapter 5).

For more in-depth information and alternative possible approaches please take a look into literature such as [8–9, 11].

2.3 Differential evolution

Next topic will be differential evolution which can be used for what we need as brain evolution – working with vectors and fine tuning of correct continuous values. This

algorithm is described in great detail in many sources, but let's sum up the basic principle and its properties. In the work *Differential Evolution Training Algorithm for Feed-Forward Neural Networks* [12] this algorithm has been successfully used for neuron training and its legibility as optimalization method is discussed and confirmed.

In our case we have separated phases of body and brain evolution. Body evolution follows already mentioned general evolutionary algorithm and for brain evolution we use differential evolution.

At the start of brain evolution phase, we take each individual in current generation and perform separated differential evolution. In this evolution we use the same body during the whole process of evolution and we change values of weights in creature's neural network which works as its brain. For more detail on creature's structure see chapter about implementation 3. For now let's just assume we have fully connected graph of neurons and we store weight setting in matrix – see figure 2.3. Here we store weights for neural network inputs, connection weights inside network and threshold values of neurons. Having this matrix allows us to unfold it into vectors of the same dimensions, which we can use in differential evolution.

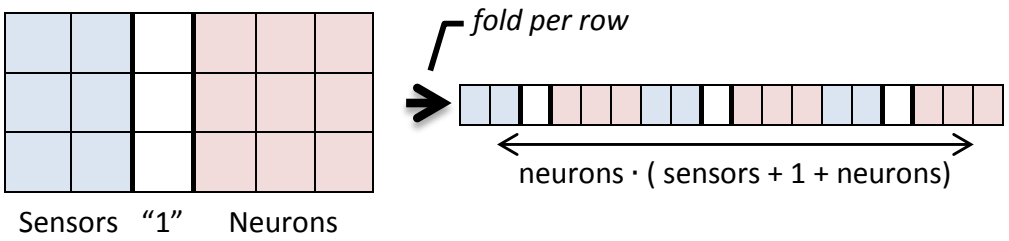


Figure 2.3. Weights setting of neural network stored in matrix.

As starting population of this brain evolution we generate random values, which are derived from currently evolved creature's initial brain configuration. In certain number of iterations we change vectors, thus matrixes of weight setting and we tune these values in order to work with current body structure the most efficiently.

One of the basic ideas of differential evolution is that we do not have any selection process – we use the same algorithm for all vectors in one iteration population.

For each vector we randomly select three more from this population – let's call them $\vec{r_1}$, $\vec{r_2}$, $\vec{r_3}$ and lets call the currently processed vector $\vec{x}$ *target vector*. Combining these three vectors in formula (3) we create new vector called *mutant vector*. We used predefined parameter **F**, the *scale factor* which is positive float number that controls the rate of population's evolution.

$$\vec{v} = \vec{r_3} + F \cdot (\vec{r_1} - \vec{r_2}) \tag{3}$$

For creation of trial vector we use *uniform crossover* (sometimes referred to as *discrete combination*) stated in formula (4) in which we use uniform random generator $rand_j(0, 1)$ for each field of vector and parameter **CR**, the *crossover probability* which determines the amount of values that are copied from newly created mutant vector to trial vector. One random index j_{rand} is always used to differentiate vectors in at least one field.

$$y_j = \begin{cases} v_j & \text{if } rand_j(0,1) \leq \text{CR} \vee j = j_{rand} \\ x_j & \text{else} \end{cases} \tag{4}$$

Finally we take this offspring *trial vector* $\vec{y}$, put it in our creature's body and test its fitness. If the fitness of this offspring $\vec{y}$ has improved upon parent *target vector* $\vec{x}$, we use it in the next iteration instead of its parent. We repeat this process for a given number of iterations and at the end, we use the best evolved brain to insert back into this creature's body. We can see the whole algorithm hinted in pseudocode in figure 2.4.

As it is with all evolutionary algorithms, there are many variations of this basic principle (as can be seen in [1]) and [13–14]), some of which are less prone to stagnation or converge faster. This is one of areas where this thesis could be built upon as is consulted in future works chapter 5.

```
1  function DifferentialEvolution(brainIterations, Creature)
2      initialize starting population X from Creature, where
3          x_0 ← Creature.brainVector
4          x_i ← rand_i(−0.1, 0.1) ∗ Creature.brainVector for i ∈ 1..PopSize
5      while iterations < brainIterations do
6          for i = 0 to PopSize do
7              select random distinct vectors r1, r2, r3 from X
8              combine v = r3 + F · (r1 − r2)                ▷ mutant vector
9              crossover y_i = v ∨ x_i                        ▷ trial vector
10         end
11         evaluate population Y
12         x_i ← BetterFitness(x_i,y_i) for i ∈ 0..PopSize    ▷ next generation
13         iterations ++
14     end
15     return best_individual
16 end
```

Figure 2.4. Differential Evolution pseudocode of implementation.

Also this brain evolution cycle often consumes more time than the body evolution. This is because during differential evolution we process all possible parent vectors and there are no selection rules active. We also tune brain of each creature in current population and we have to evaluate *popSize* of body evolution times *popSize* of brain evolution different configurations of virtual creatures.

2.4 Summary

The important things you should take from this chapter are that for the task of finding the best virtual creature structure the evolutionary algorithms are often used and are the right ones for this task. In our evolutionary algorithm implementation we used *elitism* to prevent deterioration of fitness, for selection we used transformation of fitness by order and *roulette wheel selection*. Basic evolutionary algorithm is also enriched by local search optimalization – this principle is used in *hybrid algorithms*. We are using differential evolution for brain tuning of virtual creatures. Generally genetic operators work with the mutation or crossover creation of single individual, however here we are operating with whole populations in each iteration step. The exact implementation follows in segment 3.6.

[1]) `http://icsi.berkeley.edu/~storn/code.html`

Chapter 3
Implementation

3.1 Implementation of virtual creatures

The overall idea of evolution and what fascinates us while theorizing about it is the process that starts in the absolute chaos of random initialization and transforms it into order, by finding viable solution in current system. Emergence of order from chaos. However, in most cases when we implement ideas of evolution into algorithmized programs (including this work) we have to consider our influence to the system which inherently influences how and which order will we find. That's why chapter about implementation is important.

Great part of implementing evolutionary algorithm is evaluation, in our case simulator which works as physical engine for our virtual creatures to live in. Its evaluation in a sense, that we provide it genotype of virtual creature and receive its fitness. Simulator imposes sets of rules for world where virtual creatures live. Its role is crucial. Physical engine should be close to reality, in order for our created creatures to be usable and plausible in real world. It also must not contain any glitches, because it's in the nature of evolutionary algorithms to find these and exploit them to maximize fitness. Also choice of simulator dictates the form in which we represent our creatures (or at least dictates the convertibility of our creatures into simulator's genotype). However selecting already working physical engine framework saves our time and protects us from reinventing the wheel.

From many possible engines we chose the experiment friendly program Framsticks [1]) created in Polish Poznan University of Technology for studying the virtual life. It offers us great customizability of experiments, allows us to use their scripting language in rule definitions and the basic version is free. Part of Framsticks is a client program runnable under Windows, Linux and MacOS. For visualization it allows us to use version with graphical interface seen in figure 3.1 and Theater for displaying single creatures.

It also brings us constrains of compatibility with their chosen genotype representation and the stick-natured structure of created creatures.

3.2 Framsticks representation

Framsticks provides us with multiple ways how to represent a creature, but most of them differentiate only in genotype code structure and not the actual phenotype realization. In this work we chose the so called *f0* representation, because it's the most general one (for more information see 3.3). For all Framsticks representations we can say, that creatures are stick-like constructs which use neuron objects as sensors, effectors and neural network structure as brain.

[1]) `http://www.framsticks.com/`

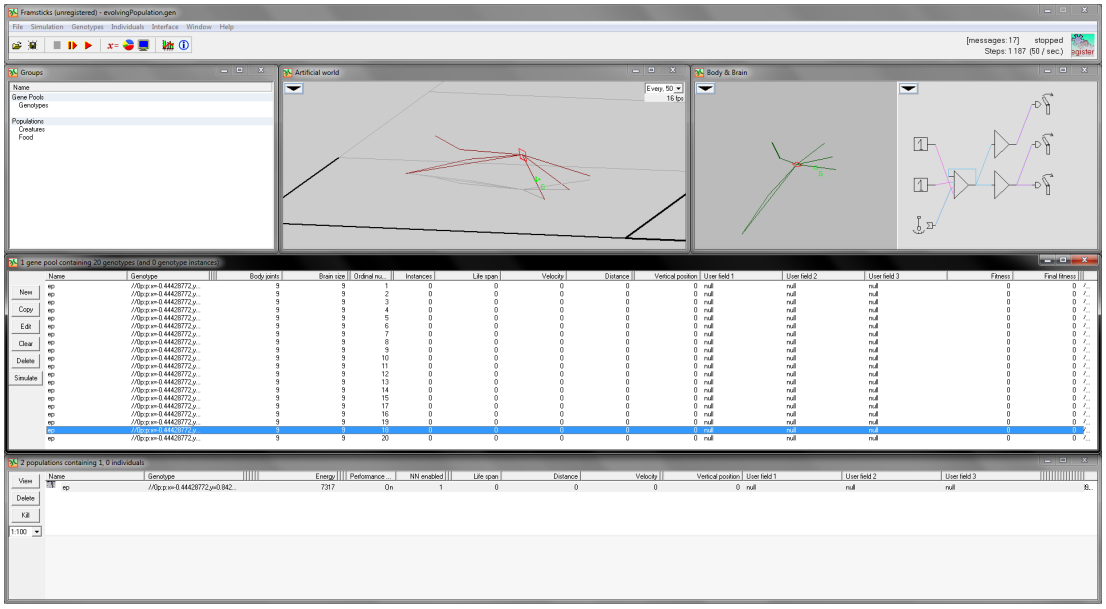


Figure 3.1. Framsticks graphical version.

3.2.1 Body representation

Body of virtual creature consists of objects of two types – parts and joints. Parts are points in space that can have weight and radius for collisions. Joints connect two parts and serve as basic edges in virtual creature's body. They have limited reach (by default 2.0 units of Framsticks measurement unit) and unlike parts they don't cause any collisions. Simple creature can be built by these components following simple rules. Each part must be connected into one component, each joint connects exactly two parts that are not yet connected by other joint and all joints are shorter than distance limit. With these rules in mind we could build creatures that are awarded in fitness by their height, which we indeed did in the first experiment (see 4.2).

3.2.2 Brain representation

For more complex creatures we need the ability of acting in reaction to their surroundings and movement of limbs. We also need brain structure consisting of neural network. In Framsticks we solve all these problems with objects called Neurons. There are many different types of neurons that vary by their effect and properties. Basically we could think of them as function black boxes with mandatory number of inputs and outputs. One of these functions is made after the artificial neuron model and is what we usually understand under the term *neuron*. Regarding their properties, the difference between Framsticks neuron objects is in their position in creature's body, number of inputs and number of outputs. Neuron location can be positioned into part – for example as a sensor function that monitors the state of certain part. It can be positioned into joint in the case of effector function that bends said joint. Or in some cases, it can be left without position.

We can implement unofficial classification into categories of *sensors*, *effectors* and *neurons*. Sensors don't need any inputs and on the contrary they provide us with monitored value as output. Multiple outputs copy the same value. Their location is dependent on their type and on what they are monitoring. Effectors on the other hand don't have any outputs, only inputs that they use to change the joint's properties in which they are positioned. If there is more than one input, value is averaged from

connected inputs. For neuron category we use only one Framsticks object, the one that is made after artificial neuron model.

In our thesis I use structure that comes from this classification. Core creature's brain is made from fully connected graph of neurons. Each neuron can have multiple inputs and one output that can be connected with all other neurons. Neural network has an additional input layer of sensors, which can be connected to each neuron. Neural network outputs are connected with layer of effectors where each effector can be connected to one or none neuron from neural network. See visualization of this structure on figure 3.2.

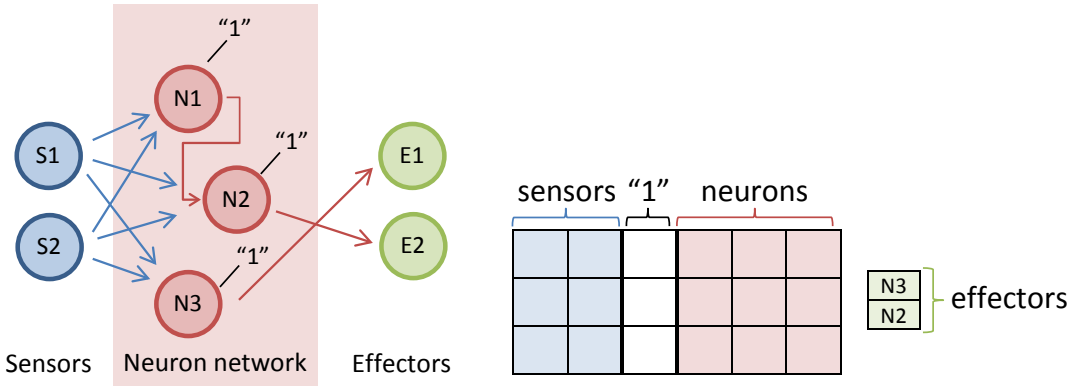


Figure 3.2. Sensor, neuron and effector structure of brain.

We can translate this entire brain structure into matrix of weights of each connection. This is an useful representation, because we can use the folded matrix as vectors in differential evolution (see 2.3). From this matrix we can also generate connections and attach them to creature's genotype.

3.3 Framsticks syntax f0

Each object is fully described by one line starting with letter that determines type of this object. This letter is followed by number of properties and their values.

```
CLASSID: Property1name=Property1value, Property2name=Property2value, …
```

3.3.1 Part object

Is described with letter `p` as `CLASSID` and in our implementation it can have following properties:

Property	Genotype name	Default value	Accepted range
mass	`m`	1.0	$< 0.1; 999.0 >$
collision radius	`s`	1.0	$< 0.1; 10.0 >$

Table 3.1. Framsticks part properties we used.

3.3.2 Joint object

Connects two parts and carries information about offset between these two parts. Properties `p1`, `p2` are mandatory and must point to reference number of parts. `CLASSID` letter is `j`.

Property	Genotype name	Default value	Accepted range
reference number of source part	`p1`	-1	
reference number of target part	`p2`	-1	
distance offset of joint	`dx, dy, dz`	0	$< -2.0; 2.0 >$
rotation in joint	`rx, ry, rz`	0	$< -\pi; \pi >$

Table 3.2. Framsticks joint properties we used.

3.3.3 Neuron object

Here it gets little more complicated, because this Framsticks object contains all possible function types, all sensors, effectors and what we usually classify as artificial neuron. Type of neuron is stored in property `d` that has value with another structure.

```
d=" NEURON_CLASS_NAME: Property1name=Property1value, … "
```

Positioning of neuron into part or joint is determined by properties `p` and `j`. These contain reference number of part or joint (see individual examples in section 3.4 or whole genotype in figure 3.5). In some cases these properties can be empty when neuron doesn't require location.

3.3.4 Connection object

It connects two neuron objects. In our implementation that means connecting sensors to neurons, neurons in neural network and neurons to effectors. Its only property is connection weight. Genotype lines for these connections objects can be generated from matrix indicated in figure 2.3.

3.4 Framsticks neuron objects

Here is a list of used Neuron objects classifies into categories by usage. In Framsticks there are more Neuron objects usable in different tasks (for example for worlds with water), see full list in here [1]).

3.4.1 Sensors

Gyroscope (`G`)

This sensor must be located in joint and doesn't have any properties. It serves as a gyroscopic sensor placed between two parts. When strictly horizontal it produces signal `0` and as it gets to vertical position it varies between range of $< -1; 1 >$.

```
n: j=1, d="G"
```

Touch (`T`)

Touch sensor is located in part and reacts to proximity of other material objects (for example floor or another part). It can be imagined as a whisker attached at the end of stick segment. Its relaxed value is `-1`, when whisker is out of reach of its range of `1.0` Framsticks distance measurement. Value `0` is reached when segment directly touches ground and greater values up to `+1` are gained when whisker is pushed into the ground by physical engine computations.

[1]) `http://www.framsticks.com/neurons_summary`

```
n: p=2, d="T"
```

Constant (*)

Doesn't have to be located in any parts or joints, only produces constant value, which can be altered by changing weight of connection that leads from this sensor into neural network.

```
n: d="*"
```

3.4.2 Effectors

Bend (|)

Bends attached joint depending on incoming signal from neural network. Bend effector rotates the second part around the first part's Z axis. It has two parameters – *power* and *range* that affect the movement velocity and can limit its movement range. On default range value of `1` the movement is unaffected and can reach full range of $< -180°; +180° >$. For visualization see figure 3.3 used from the Framsticks webpage [1]).

Property	Genotype name	Default value	Accepted range
power	`p`	0.25	$< 0.1; 1.0 >$
range	`r`	1.0	$< 0.0; 1.0 >$

Table 3.3. Framsticks bend neuron properties.

```
n: j=2, d="|: p=0.75, r=0.54"
```

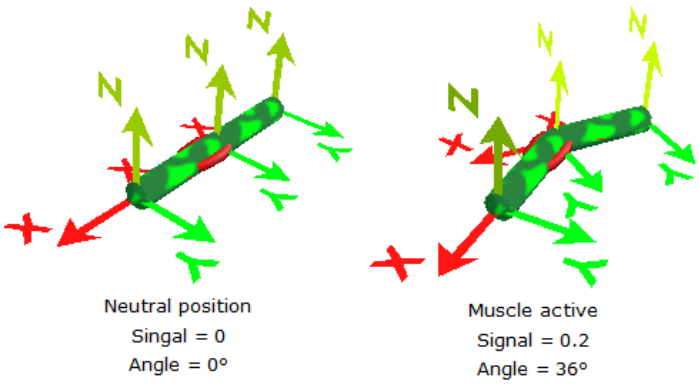


Figure 3.3. Bend neuron's effect of rotation around first part's Z axis (figure used from Framsticks webpage [1])).

Rotation (@)

Rotation effector influences the relative orientation of joint's second attached part around the first part's X axis – in a sense it twists the joint. It again has parameter

[1]) `http://www.framsticks.com/muscles_and_receptors`

Property	Genotype name	Default value	Accepted range
power	p	1.0	$< 0.1; 1.0 >$

Table 3.4. Framsticks bend neuron properties.

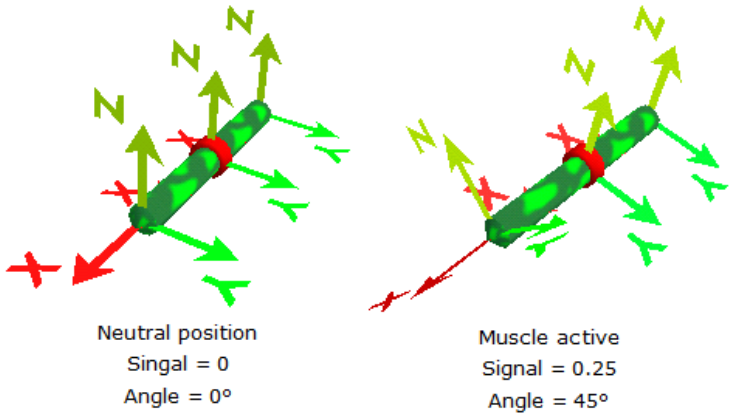
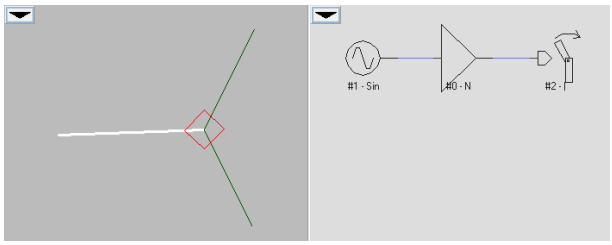


Figure 3.4. Rotation neuron's effect of rotation around first part's X axis (figure used from Framsticks webpage [1]).

```
//0
p:
p:
p:
p:
j:0,1,dx=0.75
j:1,2,dx=0.25,dy=0.5
j:1,3,dx=0.25,dy=-0.5
n:d="N"
n:d="Sin"
n:j=1,d="|"
c:2,0
c:0,1
```

genotype



phenotype

Figure 3.5. Sample virtual creature encoded in Framsticks f0 syntax.

power that influences the velocity of rotation. Rotation effector is not limited in range and can perform full 360° rotation from input signal of $< -1; +1 >$.

Rotation and bending is usually both used for relative rotation of second attached part.

3.5 Genotype abstraction

Above this f0 syntax we created our layer of representation, which can be converted into f0 and provided to Framsticks client program for evaluation. We use the structure of sensors, neurons and effectors (see 3.2) for better understanding and for better compatibility with evolutionary algorithms.

We could have worked directly with these Framsticks objects and as genetic operators use addition, mutation and deletion of entire lines, but that would bring us many difficulties while evaluating genotype. Most of these object combinations wouldn't be

[1]) `http://www.framsticks.com/muscles_and_receptors`

valid or even sensible in sense of virtual creature's structure. For that reason we used an abstraction layer that produces only valid genotype and allows us to use more complex genetic operators.

3.5.1 Joints and parts

First task was the implementation of parts and joints structure, the tricky part was to make it easily accessible for future code, mutation and crossover operations and for genotype export. We chose the representation of arrays of Parts, Joints and matrix with relationship information. The matrix simply holds Boolean indicators of connection between two parts and direction of joint. Whereas the arrays hold objects with all information on properties attached to these parts and joints. See figure 3.6.

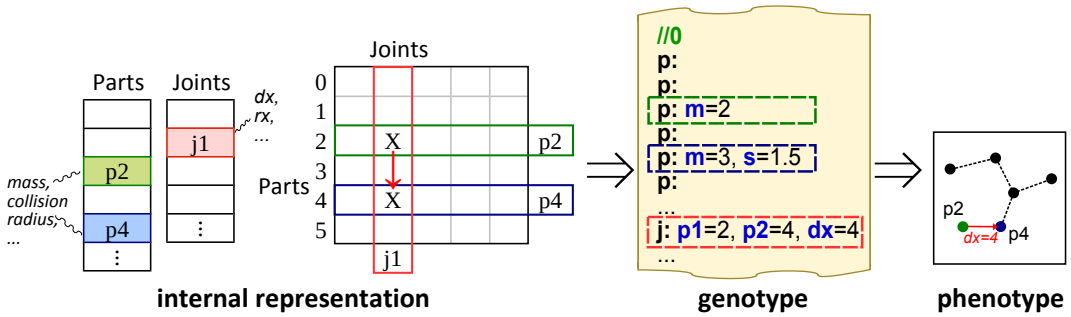


Figure 3.6. Translation of abstraction layer into f0 genotype.

3.5.2 Mutation

As mutation operations we allow addition of new objects into Parts and Joints arrays and we allow changes to interconnection in the matrix structure. Of course in changing the joint-part relationship matrix we have to follow certain rules that apply for joints in chosen genotype syntax. One joint can connect only two parts and there can't be other similar joint connecting the same parts. In our matrix that means that we can't allow the same columns. Other rule concerns parts. All parts in genotype must be connected to one main body, we can't allow more than one component in graph (see figure 3.7).

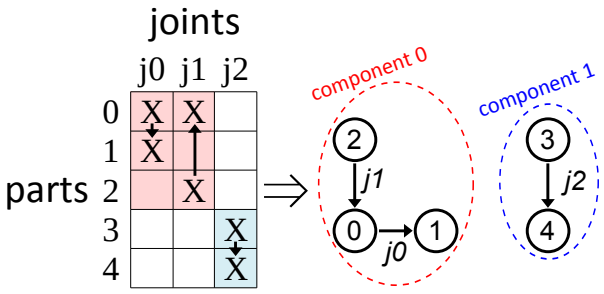


Figure 3.7. Example of more than one component in creature.

We can solve these issues either by repairing – connecting all components into one during genotype export, or we can take different approach. Only the first component (the one that contains part 0) is valid and exported into genotype. We seemingly forget about the other ones. But we also allow in our mutation operators the creation of new connections and the severing of existing connections between parts. This way we allow our creature to get rid of entire body segment which is not currently usable and

store it in temporary abstraction layer space. When this creature finds good solution for composition of its body, it can reconnect the orphaned body segment and reuse it. Properties for this whole segment (mass, collision radius and later neuron structures connected with it) are stored, because they can be used later. Of course the existence of *remove part* mutation allows creature to get rid of the whole unwanted segment part by part. In properties of joints we have stored offsets that they connect two parts with (see 3.3.2 and properties `dx, dy, dz`). During conversion to genotype these relative numbers are recalculated into absolute values and assigned to positions of parts. We start with *part 0*, giving it location of `(0,0,0)` and we go through graph and assign positions to parts. We have to watch for cycles and like in figure 3.8 we have to assign the correct values. For this we use *breadth-first* traversal method and we ignore offset values in joints connecting two parts that already have their place in x,y,z coordinates assigned. Imagine situation similar to figure 3.8, but with distance values `dx` and `dy` exactly the maximum allowed size for joint (by default 2.0). After connecting all parts we suddenly have joint of distance exceeding this maximum – therefore we have to check our creature after assigning position values and if needed scale it accordingly.

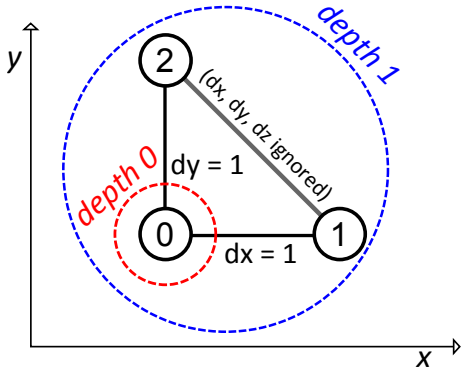


Figure 3.8. Cycle connection of joints.

3.5.3 Crossover

Next task we reached was the topic of crossover. We somehow needed to take two parents, crossover their internal representation of joints and parts and create new individual. Our first approach was inspired by what is often used in binary chain crossover, which is cutting two parents in halves (or more segments) and joining them to form offspring individuals. When we take a look at figure 3.9 we see that this leads to crossovered creatures `C` either with joint crossover (in case **a.)**), parts crossover (in case **b.)**) or both (in case **c.)**). In case **c.)** we can follow this procedure with connecting two segments, by activating some joints in areas with `0` (symbolic mark of no connections).

However this approach had bad results – we often created too large individuals with chaotically connected parts. In experiment to grow the tallest creature (when our fitness depends solely on vertical position and we don't subtract any penalties for number of joints and parts used – see chapter 4.2) it was often useful to take the most from both parents and somehow interconnect it to grow large creatures rapidly. The second and more important reason this approach is bad is because proximity and adjacency in sense of matrix (which we are cutting in halves) doesn't necessarily mean adjacency of real parts in constructed phenotype. Thus we are not cutting creature's body in half, this operation cuts parts more randomly – for example see figure 3.10.

Therefore we implemented more graph related approach, where we select components of certain depth (by *breadth-first* traversal of parts) and interconnect these by random

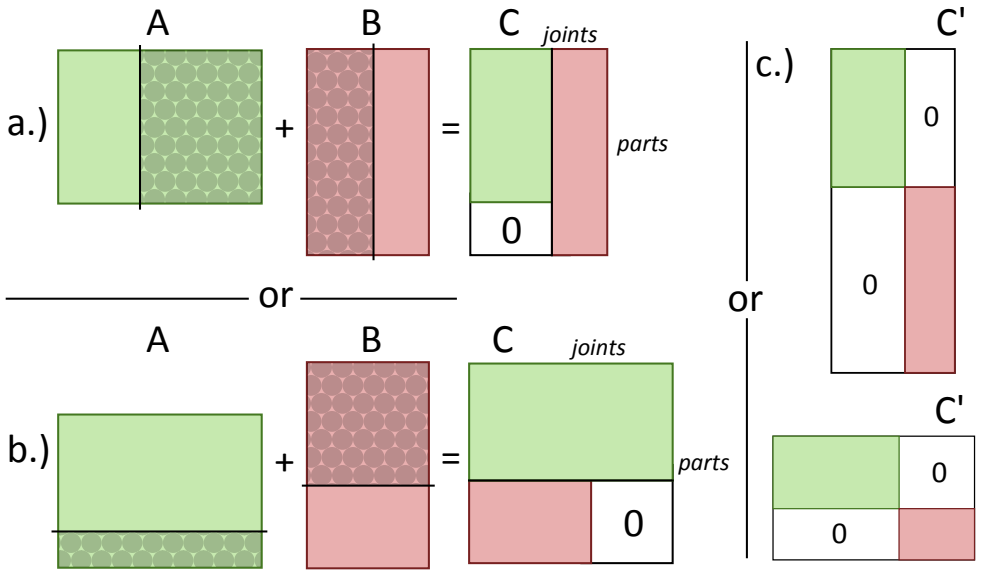


Figure 3.9. Initial crossover of parts and joints connection matrix.

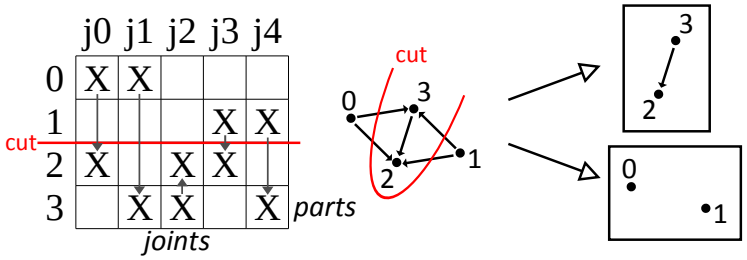


Figure 3.10. Example of incorrect crossover cutting in half.

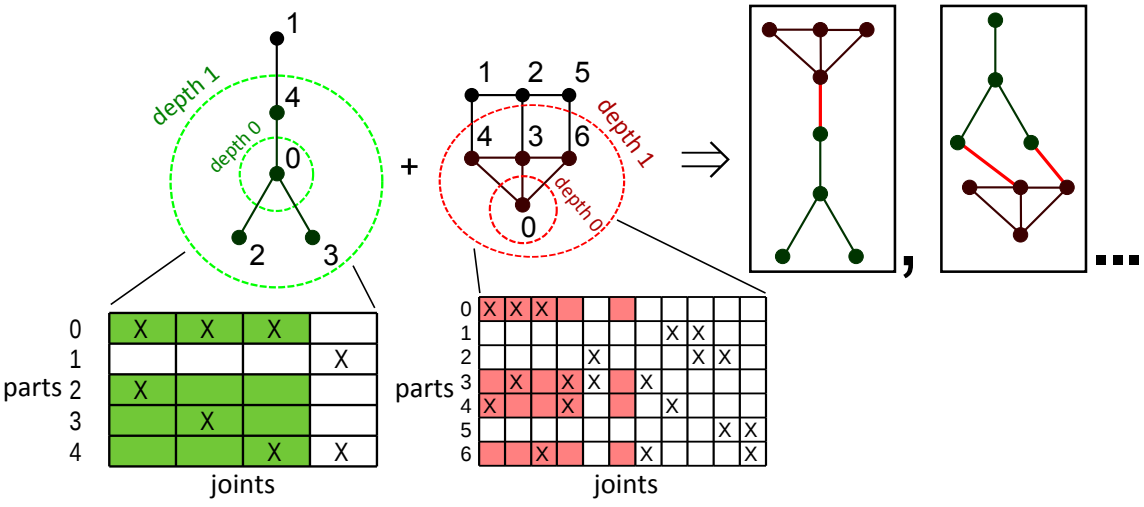


Figure 3.11. Example of subgraph crossover.

number of connections (but always at least one). As we can see in figure 3.11 this approach is resilient against chaos in referential numbers of selected parts and joints.

We can also influence how big creature we aim to create by changing the depth of selection (even though this control is limited, because small step in depth can lead to large amount of parts selected). After this selection we construct new matrix using

selected rows (*parts*) and columns (*joints*) in parent matrixes. We also use Parts and Joints objects of selected subgraph of parent's body to copy their properties (and carry for example their settings for mass).

3.5.4 Neuron, sensor, effector

Our next task consisted of implementing neural network and giving our creatures brains. This is closely connected with the second experiment, where we breed creatures capable of walking (see 4.3). Framsticks implements neuron objects, which we can classify into three usable categories – sensors, effectors and standard neurons (see section 3.4). We have already discussed brain evolution implementation and we are using the same structure as in figure 3.2. In creature's body we place several sensors to monitor its status (for example gyroscopic position of joints), which will serve as inputs for certain neuron in creature's brain. In other parts and joints we place effectors that can change joints properties and move creature's body and that take role of outputs. Standing apart from body we have fully connected graph of standard neurons that have their connection weights and individual neuron's properties set by brain evolution. From theoretical standpoint we are using differential evolution with vectors made from connection matrix of neurons (see 2.3), but from the implementation point of view its useful to think of brain evolution as of function, that tweaks all our creatures in current population of body evolution.

Implementation of sensors and effectors influences mutation operators we are using. As these are part of body, we have to allow appropriate operators of adding, removing and parameter mutation of both sensors and effectors connection. To make our task easier we implemented our brain to consist of predefined number of neurons and our body to have predefined expected number of sensors and effectors. These don't have to be connected with the neural network and similarly neurons don't have to be connected with each other (however they often end up being all connected). Instead we allow these connections from sensors and effectors into neural network to emerge during body evolution by correct mutation operators. Internal representation and interconnection of neural network can be left solely upon differential evolution, we only determine which inputs and outputs are used and where.

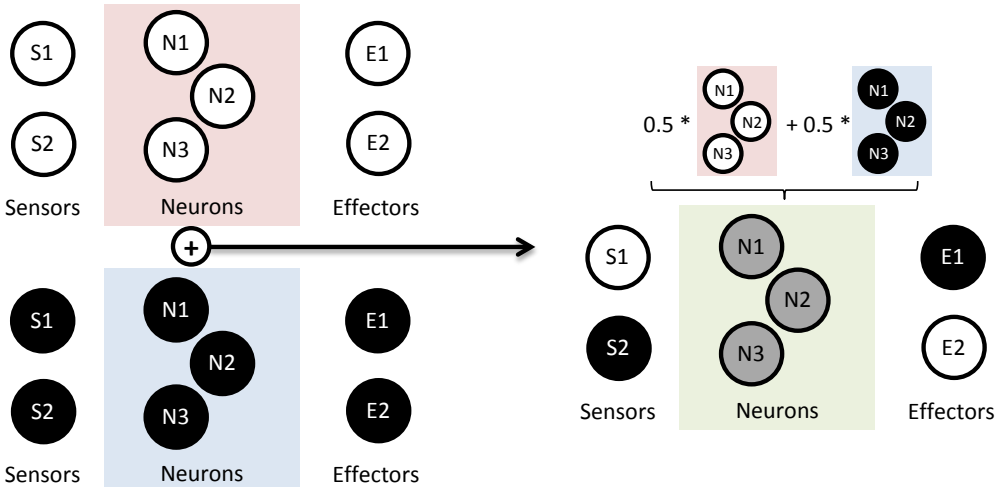


Figure 3.12. Crossover method used for sensors, neurons, effectors.

More difficulties arose with crossover – we have to try to carry functionality of sensor and effector constructs used in parents into their offspring. As sensors and effectors are

located inside parts and joints, we could carry them over into offspring with selected parts and joints (selected by subgraph method 3.11), however that would lead to unwanted sensor and effector multiplication. Instead we used the unchanging nature of neural network's dimensions and also the limited number of used sensor and effector objects in creature's body. From parents we crossover the whole sensor-neuron-effector structure as we demonstrate in figure 3.12. We *uniformly crossover* (in another term we use *discrete combination*) sensors and effectors and determine offspring's neural network weight matrix by combining matrixes of parents.

3.6 Evolutionary algorithm implementation

It has already been said in section 2.1 that we decided to split body and brain evolution. Body evolution phase is done through altered generic evolutionary algorithm and brain evolution phase with differential evolution. This combination of evolutionary algorithm and optimalization method can be called hybrid algorithm 2.1.1. For better understanding we provide pseudocode 3.13 that accompanies scheme from figure 2.1. It references other functions with their pseudocode – `DifferentialEvolution` in 2.4 and `BodyEvolution` in 3.14.

```
function Evolution(bodyIterations, brainIterations)
    initialize startingPop
    evaluate startingPop
    parentPop ← startingPop
    while !EndCondition do
        for iteration = 0 to bodyIterations do
            parentPop ← BodyEvolution(parentPop)
        end
        foreach Creature ∈ parentPop do
            Creature ← DifferentialEvolution(brainIterations, Creature)
        end
    end
end
```

Figure 3.13. Pseudocode of used evolutionary algorithm.

3.7 Framsticks world definitions

Framsticks as a physical engine and evaluator gives us great variety in world rule set definition. We can use the basic referential code and only slightly adjust one variable if we need to, or we can write our own entire scripts in Framsticks scripting language *FramScript* for custom evaluator behavior (for more informations on *FramScript* visit [1]). For our needs we can sufficiently use the preset experiment definition file `standard-eval.expdef` and change only couple of variables. The most important change here is the choice of parameters that are used for fitness computation and their individual weights. For our experiments we needed at first to evaluate height of creature (vertical position) and somehow include the number of used parts and joints (in experiment 4.2). In our next task, we wanted to evaluate the distance traveled by creatures over their lifetime (in experiment 4.3).

[1]) `http://www.framsticks.com/common/script/framscript-lang.html`

```
function BodyEvolution(parentPop)
    order creatures by fitness
    offspringPop = []
    while |offspringPop| < explorationSize do
        select one method:
            - mutation:
                select one creature with roulette selection
                mutate creature
                return as offspring
            - crossover:
                select two creatures with roulette selection
                crossover creatures
                return as offspring
            - crossover + mutation:
                select two creatures with roulette selection
                crossover creatures
                mutate crossovered creature
                return as offspring
        add offspring to offspringPop
    end
    evaluate offspringPop
    selectedPop = []
    selectedPop ← first | Elitism | best creatures from parentPop
    selectedPop ← first | PopSize − Elitism | best creatures from offspringPop
    return selectedPop                    ▷ as parentPop for next iteration
end
```

Figure 3.14. Pseudocode of used evolutionary algorithm for body evolution.

We must keep in mind that the manner in which Framsticks computes creature's fitness, can determine the shape and structure of created creature quite significantly. This can result into creatures cabaple of living only in this simulation environment and not being viable for real world assemly. For some information we can look on Framsticks webpage [15] or in appropriate chapters in [16].

Framsticks also allows the usage of multiple physical engines. This choice is also made in experiment definitions file. First possible choice is their simple internal engine called *Mechasticks* (its processing of body simulation can be seen here [1])). The second engine is called *Open Dynamics Engine* (an open source library for simulation of rigid bodies [2])) and was implemented into Framsticks with the hope of evolving more credible creatures with engine closer to reality then the internal one. We used the second one for the same reason and also because in *Mechasticks* large creature structures crumbled under their weight (thus not allowing to breed creatures taller than certain height in the first experiments).

In this experiment definition file we can also set several parameters that influence the speed of evaluation. If we used non-deterministic noise neurons, we could repeat evaluations for certain number of iterations and get fitness averaged. We also determine the lifetime of created creatures which determines how long the simulation is running.

[1]) `http://www.framsticks.com/a/al_simdetail.html`
[2]) `http://ode.org/`

Chapter 4
Experiments

4.1 Experiment preparations

With the implementation explained in previous chapter, we can try our algorithm on several experiments and watch its behavior.

4.1.1 Referential settings

As the world of possible solutions for valid virtual creatures is highly multidimensional we had to focus on one parameter per experiment. By selecting one parameter and trying different values, we perform cutting through this multidimensional world in direction of one dimension. We can't allow ourselves to change several parameters at once, because our results would be clouded and ambiguous. Therefor we implemented referential settings, which consist of default values for all parameters and we alter only one at time. We chose these so computation ends in sensible time and we can collect results. These parameters also provide us with the list of possible experiments we can carry out.

parameter	description	ref.value	exp.
bodyIterations	number of repetitions of body evo.	2	4.3.4
bodyPopSize	population size of body evo.	8	4.3.5
bodyExpSize	explored individuals in body evo.	$2 * bodyPopSize$	
bodyElite	elitism	1 (minimum 1)	
brainIterations	number of iterations of DE	2	4.3.4
brainPopSize	population size of DE	6	4.3.5
neuronNum (N)	used neurons in neural network	4	4.3.3
sensorNum (S)	inputs to neural network	4	
effectorNum	outputs from neural network	4	
F	DE *scale vector*	0.8	
CR	DE *crossover probability*	0.1	
dimension	DE *dimension*	$N * (N + 1 + S)$	

Table 4.1. Referential parameters for experiments. Here we are using abbreviations DE for *Differential Evolution*. *Dimension* is dependent on number of neurons and sensors used and can't be changed as other parametric values.

4.1.2 Experiment execution

In experiments we chose one of the parameters from 4.1 and we varied it in sensible range. We wanted to see extreme behavior, when the parameter is set intentionally imbalanced with the rest of the setting to see how it fares. We also wanted to know what the most optimal settings would be and how important this parameter was in comparison with others. We repeated each experiment and each parameter value over

100 instances for predefined number of iterations. The results were averaged over 100 instances. We also took the final iteration of all instances and we used this final data in several whisker box plot graphs. This is done with the idea, that in real life scenario we don't really care how the creature evolved, we will only use the final iteration's best creature.

We will try to comment the behavior of experiment, provide explanation for it and compare it with other instances. We will show several graphs and reference the others that manifest the same behavior. All interesting graphs will be attached in appendix B.

4.2 Experiment 1, the tallest creature

4.2.1 Experiment summary

We started with the experiment of breeding the tallest creature. For this we don't need brain evolution and in fact we used this experiment as framework for our algorithm during development. These results were of course calculated by the final version. As we only need creature's body structure to grow, we can disable the brain evolution phase and forbid all neurons, sensors and effectors. We expect our creatures to grow in building like structures that are balanced and can carry tall segments.

We can also expect our creatures to grow infinitely. If there isn't any roof for creatures to hit in number of parts or joints, they will grow into great structures. In order to prevent this hungry approach (always create new parts and joints to attach on virtual creature), we can alter the fitness function. To the basic positive evaluation for vertical position of creature, we can add penalizations for number of used building blocks (joints and parts). This will force the evolution to create space efficient creatures – taller and thinner than without this rule. In some cases we could encounter negative fitness (creature that has fallen over with too many joints and parts), but our algorithm is resilient to this, because it transforms fitness by rank (see 2.2.1). For our graphs to look nicely, we chose to set some small fitness offset. Also we didn't alternate between parameter settings in this experiment as it was only the first step for next experiment.

4.2.2 Experiment definitions

In experiment definition file we must set appropriate variables for fitness evaluation (first 4 rows in 4.2).

parameter	wanted value	expdef var.
fitness for vertical position	1	`ExpParams.cr_vpos`
fitness for number of parts	-0.035	`ExpParams.cr_gl`
fitness for number of joints	-0.035	`ExpParams.cr_joints`
constant fitness offset	0.5	`ExpParams.cr_c`
physical engine	1 (*ODE*)	`World.simtype`
lifetime value	50	`ExpParams.Energy0`

Table 4.2. Experiment definition for breeding tallest creature.

For this experiment it's also important to set the physical engine to *ODE*. Because we are not simulating movement in this experiment, we can spare some computational time by setting the lifetime of individuals low. The simulation starts after the creature

stabilizes in initial balanced position, so the eventual falling over creatures get low fitness regardless on set lifetime value. We can also turn off all neuron evaluation, but because our generated creatures are already constructed without them, the performance gain is minimal.

4.2.3 Results

As we didn't change any parameters in this experiment, the results are straightforward. We repeated the same experiment 100 times and got resulting graph in figure 4.1.

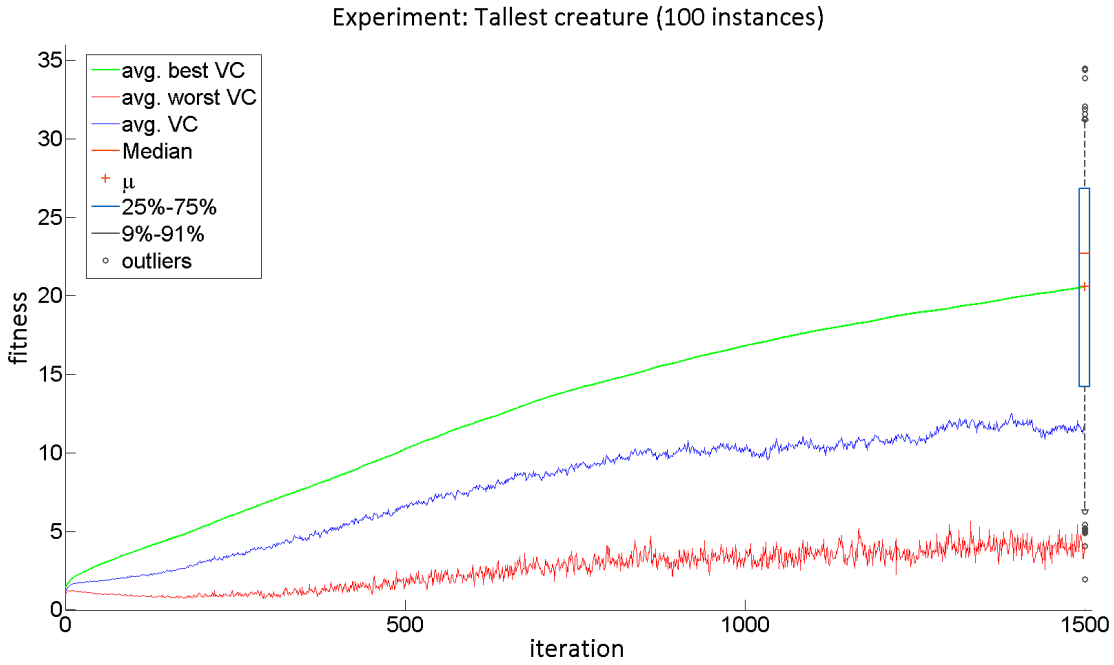


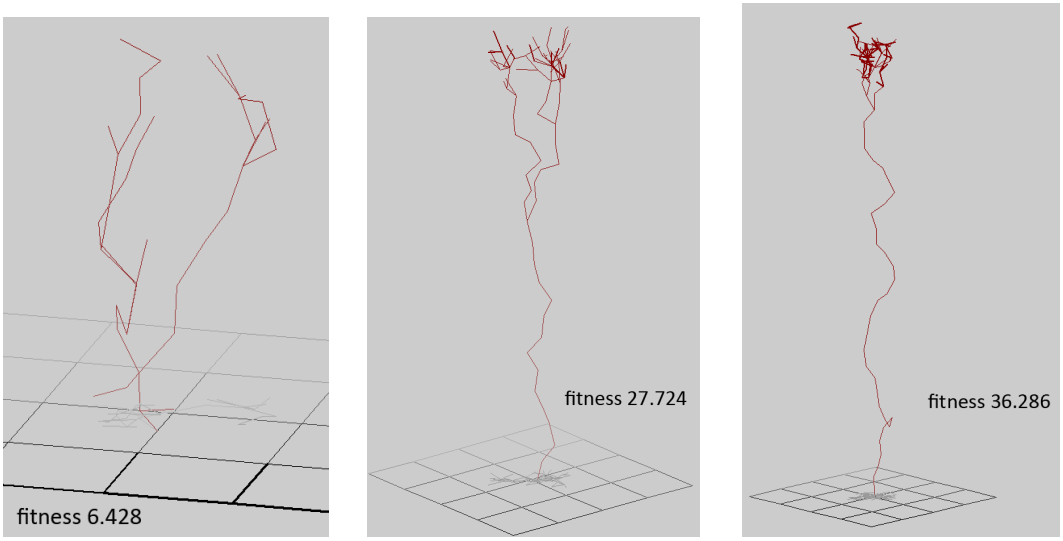
Figure 4.1. Results of experiment 1, breeding the tallest creature (in graph legend we are using abbreviation VC, which stands for *Virtual Creature*).

We can see that the fitness rises over number of iterations, which is consistent with our expectations. To grow tall creature should be a simple task, achievable even with simple mutation and crossover rules. Also the fact that we penalized creatures by number of joints and parts didn't endanger the growth. We could have encountered some kind of stagnation in situation where the possible growth in vertical position would be always counterbalanced by loss in fitness for newly attached joints and parts. That would lead to maximal possible fitness value and even possibly in existence of global extreme. However with settings of parameters following table 4.2 this hasn't occurred.

4.2.4 Sample individuals

When we take a look at individuals produced by this experiment, we notice that the majority of them looks like creature **c.)** in figure 4.2. They are indeed tall and they seem to be balanced on one leg only. This is quite surprising – it's due to the manner in which Framsticks calculates vertical position fitness. It takes vertical position of all parts and joints and uses the weighted average as vertical position of creature. So it's advantageous for our creatures to concentrate as many parts and joints as is possible into upper body segments and gain grated calculated vertical position. This case is an

example of physical engine dictating and influencing the shape of cultivated creatures. However this approach has its disadvantages, because of situation illustrated in figure 4.3. Here we can see prosperous individual that is still balanced and one of its offspring, which has lost this balance and has fallen and received significantly smaller fitness. This could explain the large gap between the best average individual and the worst average individual in figure 4.1 – it's common for individuals of this experiment to lose their balance and fall. However we can see that this doesn't stop the evolution and the growth itself in figure 4.4. Perhaps for purposes of balance, we can often notice the emergence of multiple branches at top of creatures – similar to rich treetop.



a.) b.) c.)

Figure 4.2. Sample experiment 1 creatures.

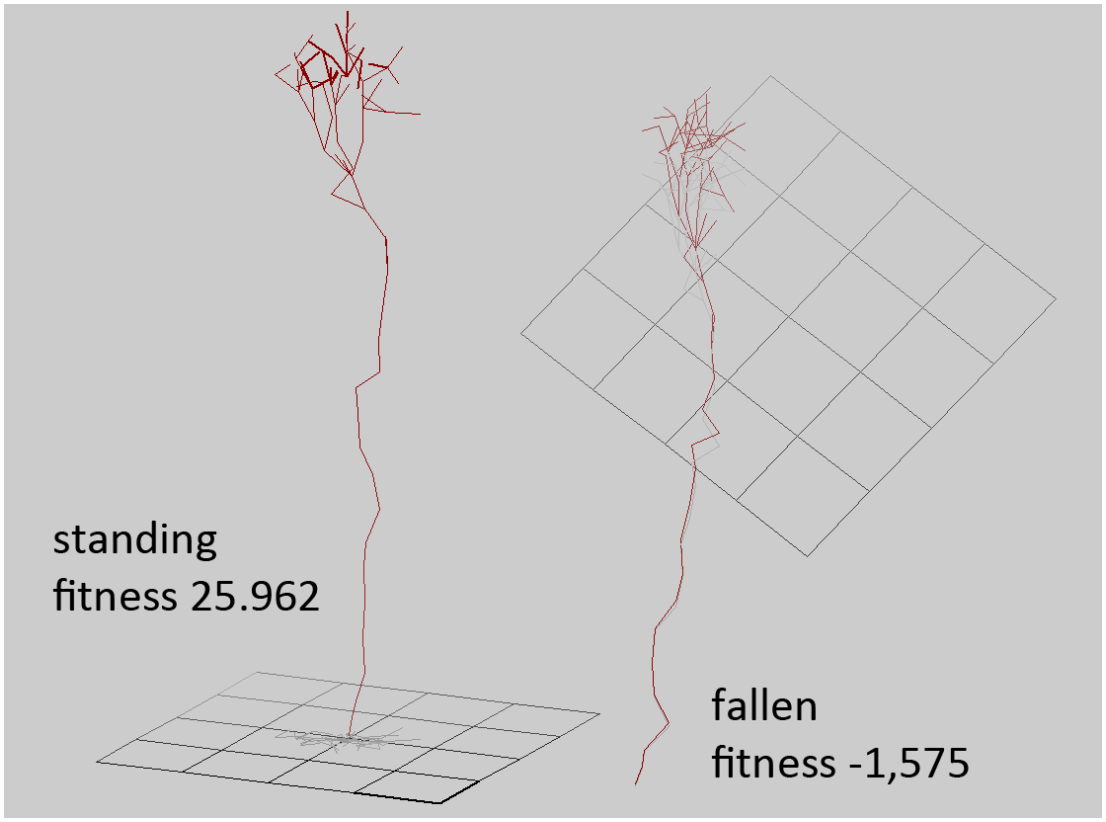


Figure 4.3. The problem of balancing out creatures. Fallen offspring will have significantly lower fitness.

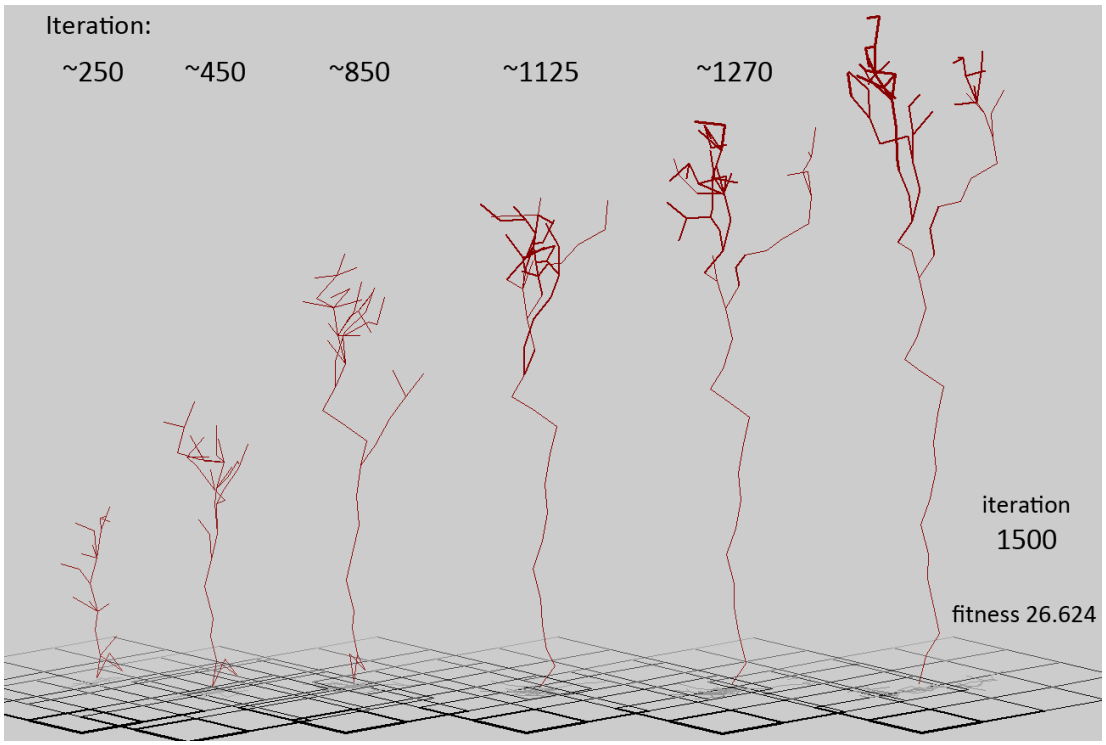


Figure 4.4. Sample experiment 1 growth with iteration numbers. Creature grows its tree-top and relocates most parts and joints to its upper segment.

4.3 Experiment 2, the fastest creature

4.3.1 Experiment summary

The second experiment is also heavily inspired by Karl Sims walking creatures. Unlike in the first experiment, we now need brain evaluation and the usage of neurons, sensors and effectors. We also tried changing several parameters in order to study this experiment more thoroughly. Our fitness is now computed from distance creature manages to move from initial position. We decided not to simply measure average velocity to prevent the occurrence of creatures moving in circles. If we measure distance passed over the whole lifetime, then the ones that move in one direction are more likely to obtain greater fitness. Their movement speed is used more efficiently in surpassing distance.

We expect multiple ways of reaching this criteria to occur as movement can be performed in various manners. We also expect emergence of movement in one direction without much steering, as that would decrease passed distance. In bodies we inherently anticipate usage of sensors, neurons and effectors constructs as effectors are the only way to make creature moveable. The alternating phases of body and brain evolution should complement each other and allow tweaking of weights of neural network for created creatures. With limited number of iterations we can also expect some configurations of parameters and random initial starting populations to do poorly and not develop satisfactory method of moving. We chose several parameters for experimentation and we discuss their specifications with their results.

4.3.2 Experiment definitions

This time we only need one parameter evaluated for fitness, however we need to have it processed with large delay, as we want to count the distance over whole lifetime. In Framsticks experiment definition file we can arrange this by setting the `perfperiod` close to the value of `ExpParams.Energy0` (lifetime of creature in energy). This parameter influences how often is called the performance evaluation and thus setting it near the moment when creature's lifetime is nearly over, we get the distance traveled from starting position. Also setting longer lifetime in this experiment allows us to test creature's ability to walk on larger area. If we had it set up for too small value, creatures, that only initially fling themselves and then remain still would be awarded with high fitness. Therefor we had to increase this value even though the evaluation time of each creature takes more time and the overall performance is slower.

parameter	wanted value	expdef var.
fitness for distance	1	`ExpParams.cr_di`
performance period	950[1])	`Population.perfperiod`
lifetime value	1000	`ExpParams.Energy0`

Table 4.3. Experiment definition for breeding fastest creature.

4.3.3 Results – alternating neuron number

In the first variation of this experiment, we chose to alternate the number of used neurons. We keep the same amount of sensors and effectors, in attempt to explore

[1]) Value of 950 is used as number which is close to creature's lifetime. We didn't want to risk the emulation ending before performance evaluation, so we set it to a value little lesser than lifetime.

the behavior of only one parameter (see figure 4.5). We expect this change to have significance in resulting complexity of creature's brain as it is fully connected graph of neurons. Of course the higher number of used neurons raises the dimension of explored space in differential evolution. This fact has two effects. The computation takes longer and we can expect some difficulties of finding the correct solutions in vast area of multidimensional space. Theoretically the higher count of neurons should mean more complex brains and behavior, but the question is, whether these solutions will be found in time period limited by number of iterations. Body and brain iterations are both repeated twice, until they reach the ending 500th iteration, where the algorithm is stopped. Both phases get their 250 iterations and therefore should be balanced.

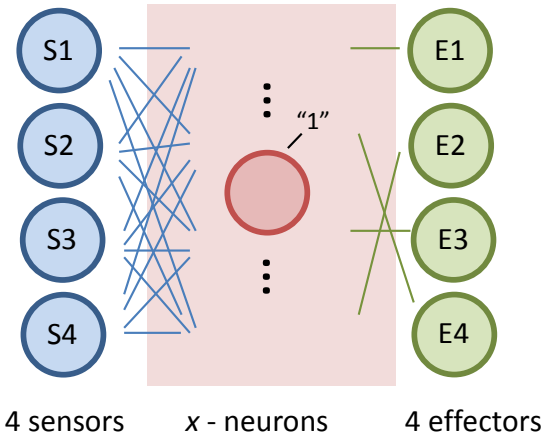


Figure 4.5. Alteration of neuron number $x \in 1 \rightarrow 50$, the sensor and effector number remains the same.

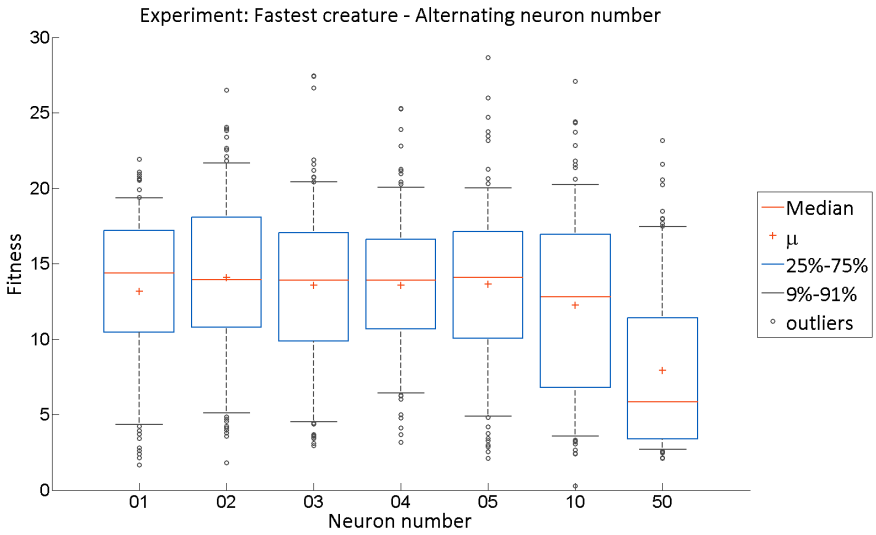


Figure 4.6. Whisker box plot graph comparison of all tested values. Neuron number values are both detailed at first and extreme in case of 50 neurons.

When we take a look at results in figure 4.6, we may notice, that movement is clearly achievable with all configurations of brain. However when we see the comparison of individual fitness growth of for example configuration with 2 and 10 neurons in figure 4.7, we can see the difference. As expected, the success rate of configurations with

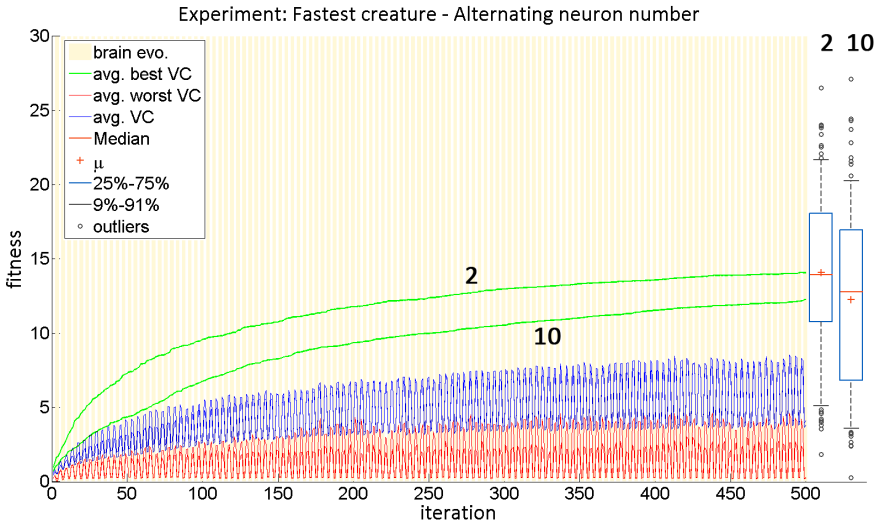


Figure 4.7. Comparison between virtual creatures evolved with the limitation of neural network size – 2 and 10. VC stands for *Virtual Creature*.

higher neuron numbers (10 or even greater extreme 50) is lesser. It seems that it's more difficult to find the correct weights matrix with higher dimension of problem. If we balanced this by increasing population size or by extending the number of iterations, we would possibly see more complex creatures to emerge and catch up or even surpass the simple ones.

4.3.4 Results – alternating iteration ratio

In the second variation we tried altering the iteration number of body and brain evolution phases. This number represents for how many times each phase is repeated before switching to the other one. The combined number of iterations remained the same – after 500 iterations, the algorithm was stopped no matter in which phase it currently was. By changing the ratio between these numbers, we are alternating the balance of attention placed on each phase. Also changing the repetition number from referential 2 to for example 50 means, that there will be lesser number of switches between the two phases, which could be disadvantageous. We expect to see how much does this balance of brain and body evolution actually matter. For parameter values we chose 4 numbers in range of 1 to 50 – 1, 10, 25, 50. We chose this range to test extreme values like 1 or 50 and also the more balanced ones as 10 and 25. This gives us 16 combinations of configurations to be tested. For clarity purposes we chose to visualize four values in each graph – displaying the cut of one parameter through multidimensional space and having the second parameter set on one value.

When we take a look at results in figure 4.8, we can notice, that with growing brain evolution iterations number, we receive better results of best individual in final iteration. We can see similar, yet less significant growth in figure B.2. However when we compare these results with the ones in figure 4.9, the witnessed behavior is different. With higher number of brain iterations the overall performance of fitness is lowered. To explain this behavior we must inspect the individual growth of the configuration of 1 body and 50 brain iterations in figure 4.10.

As we can see, the phases are extremely imbalanced – in this case the body evolution phase occurs for only 9 iterations spread apart in graph. The same applies in other, less extreme cases. There must be maintained balance between phases, otherwise no

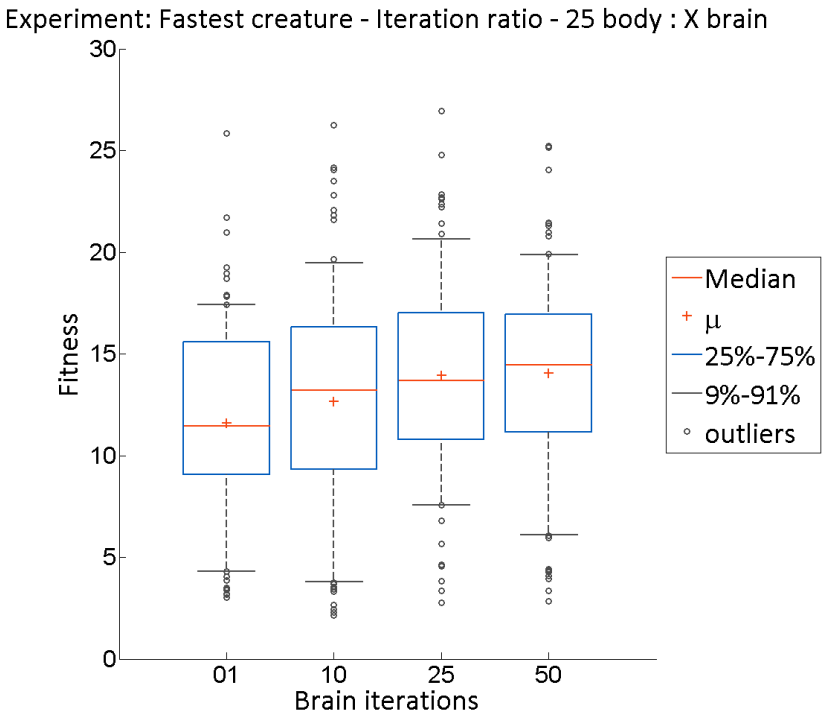


Figure 4.8. Comparison of alternating brainIteration number with bodyIteration of 25. Appears well-behaved and consistent with theory in 4.3.4.

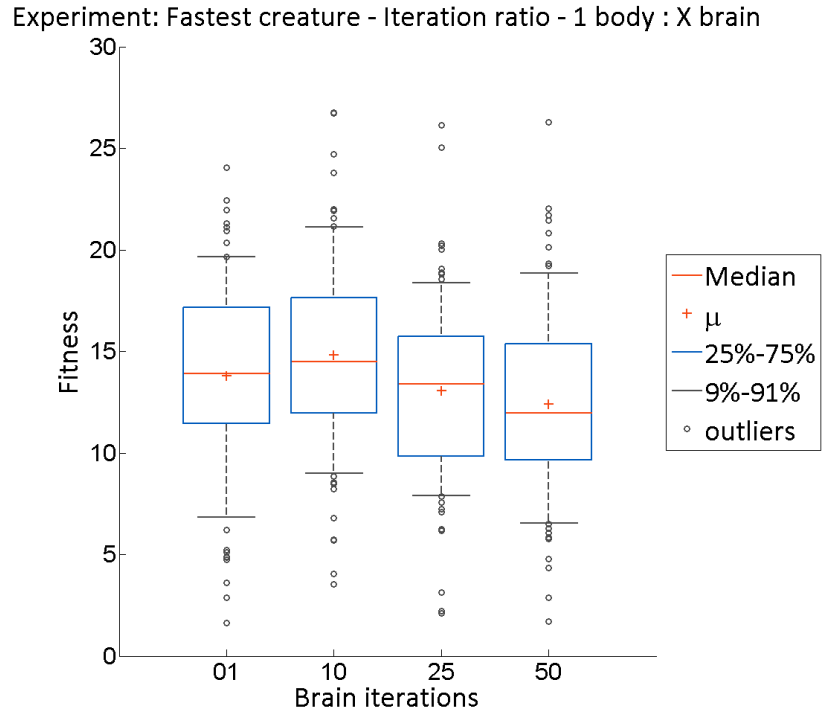


Figure 4.9. Comparison of alternating brainIteration number with bodyIteration of 25. Appears ill-mannered and inconsistent with theory in 4.3.4.

matter how good the brain evolution is, it can't do much with badly constructed body.

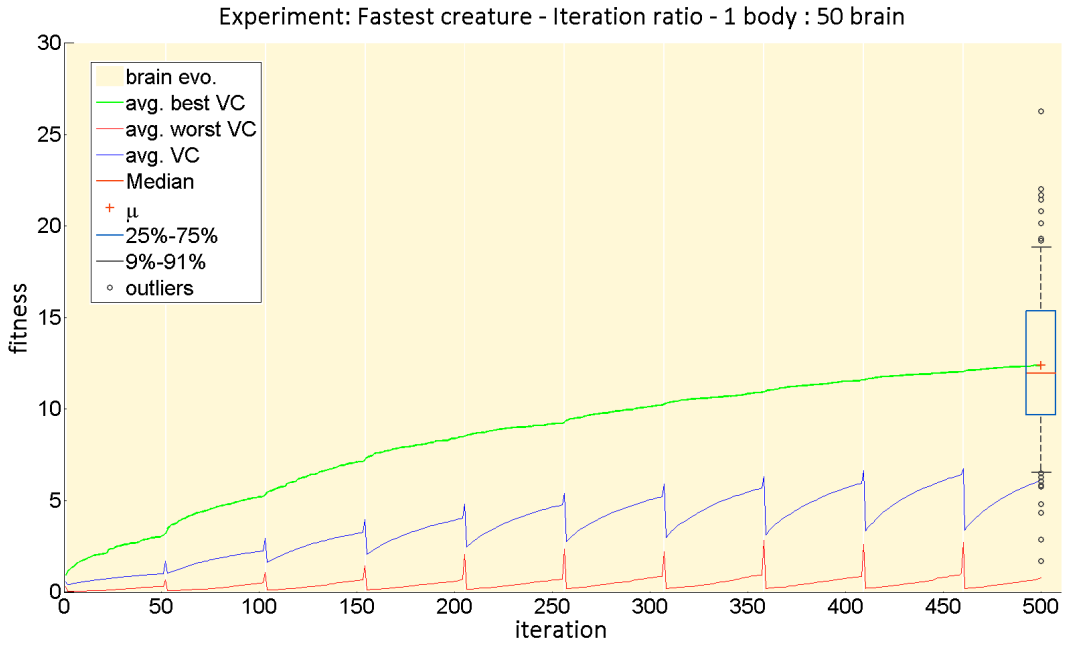


Figure 4.10. Sample A – graph of evolution with extremely unbalanced iteration ratios. Yellow highlighted area corresponds to brain evolution phase. Notice that number of body evolution phase is very low. VC stands for *Virtual Creature*.

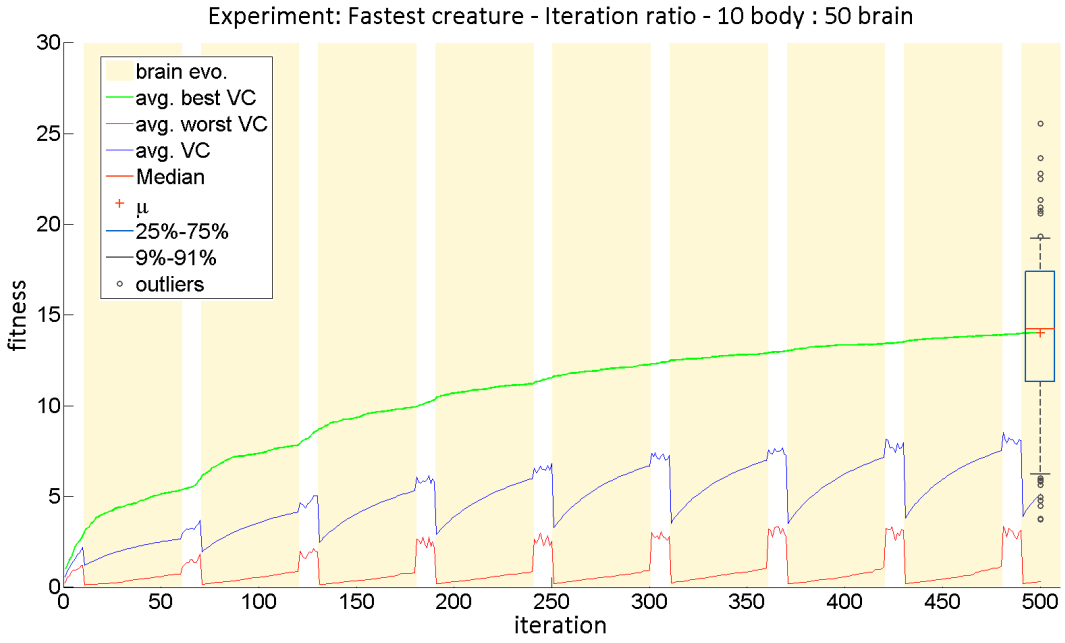


Figure 4.11. Sample B – graph of evolution with slightly less unbalanced iteration ratios. Even small change influences results. VC stands for *Virtual Creature*.

We can see, that even small step, when we chose 10 body iterations instead of just 1 makes significant difference in figure 4.11.

The best balance is achieved when the brain and body iteration number is the same – as it was for example in our referential configuration. With greater and balanced values, we can say that the number of switches between phases is lowered and it reflects

in performance. If the iteration wasn't limited by 500, we could see better results from these configurations, where tweaking in each phase is done more thoroughly (for more iterations).

The other extreme we could watch is on the other end of spectrum, with 50 body and only 1 brain iteration in figure 4.12. Again the overall performance is not very good in comparison with the more balanced ones. Also notice the apparent jump in fitness after only one brain evolution iteration. It appears that the body is good enough for even small brain tweaking to matter.

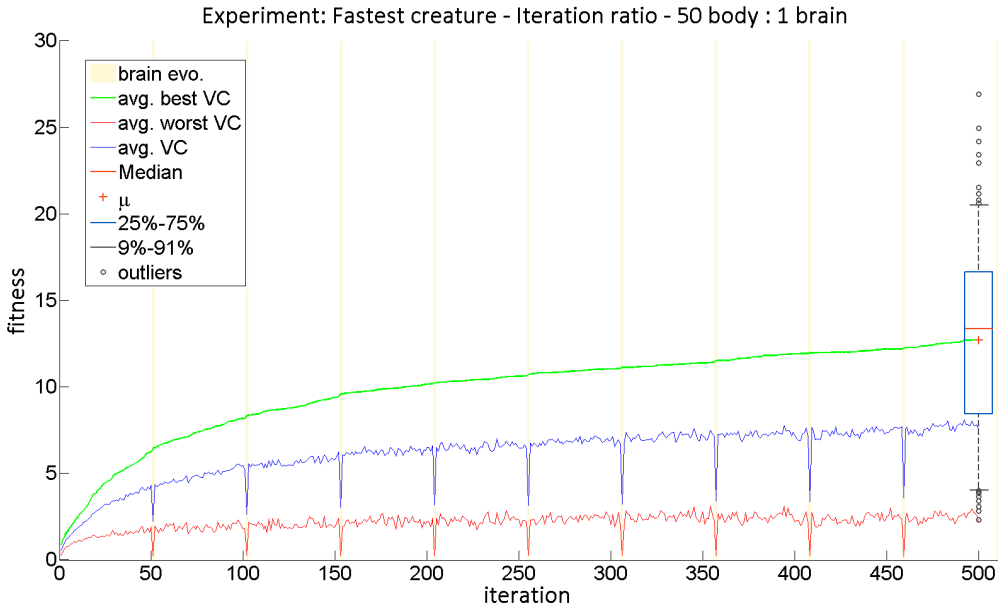


Figure 4.12. Sample C – graph of evolution with the other extreme – 50 body iterations and only 1 brain iteration. VC stands for *Virtual Creature*.

4.3.5 Results – alternating popSize ratio

In the third and final variant we focused on parameter of population size in both phases. Body *popSize* determines number of individuals, whereas brain *popSize* corresponds with the number of tried brain configurations for each individual in each iteration of brain evolution. Again we will change this ratio and try to determine its effect on experiment. If we can imagine the previous variation of number of iteration as changing the length of explored space, then changing the population size would be like changing the width of this space. In a sense both variants influence the number of explored individuals, only each one does it in different manner. We can expect the experiment to show growth with higher population size. In this case, the chosen value doesn't influence the balance of body and brain iteration ratio, so higher values should always be better. Of course in the aspect of computational and time requirements, higher population will result in slower run of algorithm. Body *popSize* directly influences even the brain evolution phase, because in differential evolution we tweak brain for each individual in population. This experiment will probably be more interesting in comparing the importance of body and brain phase (we already know from previous variants, that having them both implemented in beneficial).

This time, we used parameter values of 5, 10, 15 and in special case 20. The population size in referential solution is 8 for body *popSize* and 6 for brain *popSize*. As we

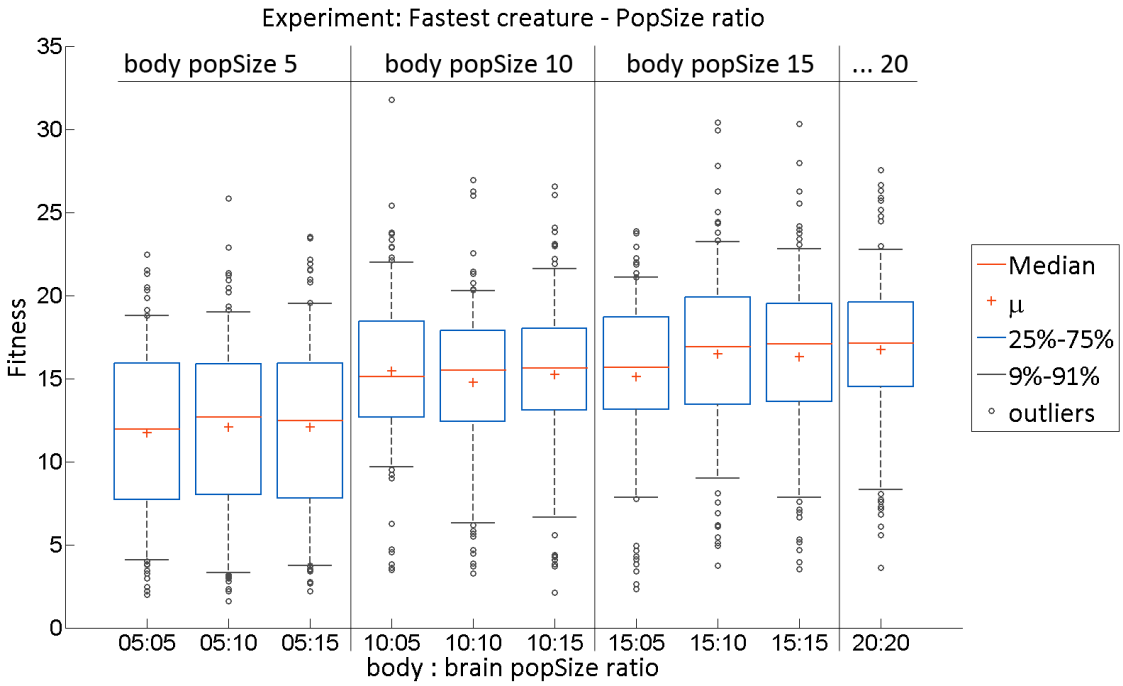


Figure 4.13. Comparison of various population size ratio in body and brain evolution phases. Growth in body evolution popSize seems to be more significant.

have fewer combinations, we can fit them all in one graph. We separated changes of parameter body *popSize* by lines in figure 4.13.

We can clearly see that the increase in population size of body evolution is more significant than the increase of brain population size. Changing *popSize* of brain evolution phase doesn't really have that significant effect. For example with body *popSize* of 5 the brain *popSize* almost doesn't matter. We could say that having correctly constructed body is the prerequisite for optimal function of brain evolution. We can imagine the situation, where we only have incorrectly placed sensors and effectors and no matter what we do in neural networks, we can't make it work. We also could say that the reason for increased significance of the body population size parameter is due to its usage in determination of exploration size (which is *bodyPopSize * 2*). Exploration size is used solely in body evolution and it affects number of explored individuals. After each iteration we keep only the first best *bodyPopSize*. Also the number of body population size matters when determining what individuals we try to tweak in differential evolution. To confirm these finding we explored even the combination of 20 body and brain population size. We can witness, that the gained increase in fitness is not as significant as is in the case of jumping from 5 to 10 body *popSize*.We didn't exhaust all possible combinations, because it is quite computationary expansive. The same applies to higher values of population size. For example in differential evolution of experiment with 30 body and brain *popSize*, we would need to compute $30 * 30$ individuals in each brain evolution iteration.

4.3.6 Sample individuals

In attached figures we selected some interesting manners of movement, which was acquired during evolution. For convenience the commentary is always placed in figure's description.

Video presentation of the best individuals is also available on `http://www.youtube.com/watch?v=qRhW1NPRn0g` or for convenience stored on attached CD.

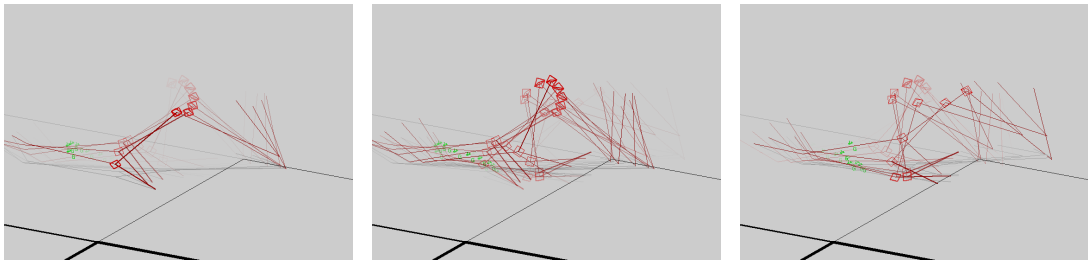


Figure 4.14. Attempt to capture creature's movement pattern. Creature walks with the help of alternating contraction and relaxation of leg-like construct. This creature's movement is in a way similar to inchworm's method of walking.

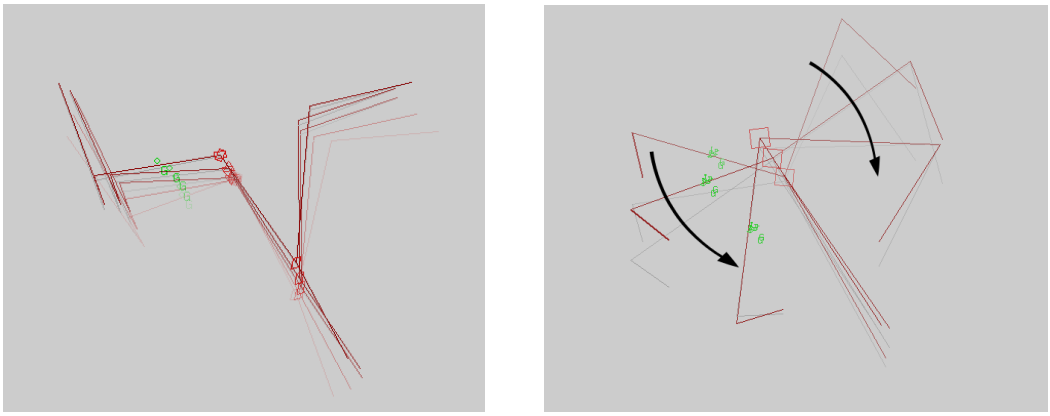
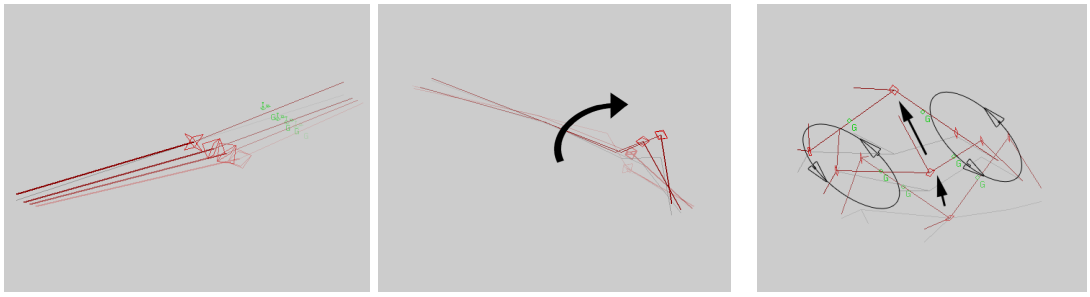


Figure 4.15. Crawling and synchronization of limbs. Several creatures have resorted to crawling with the help of synchronized limbs, which push the whole creature forward.



a.) b.) c.)

Figure 4.16. Samples with rolling movement. Creature **a.)** adopted very efficient movement in surprisingly simple body. It has both bend and rotation effectors in the middle intersection of joints and uses slight bending accompanied with rotation to exhibit wheel-like movement. Creature **b.)** has slightly longer body and rotates one ending leg segment. Creature **c.)** has developed synchronous rotation movement in both legs and it rotates very efficiently it's whole body.

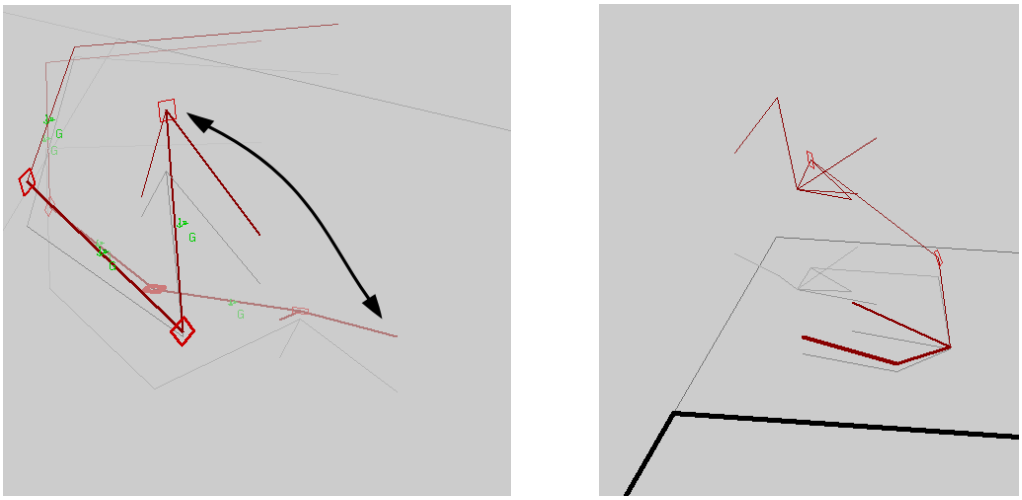


Figure 4.17. Pushing and jumping. Several creatures use the method of contracting and using stored potential to jump like spring. Sometimes this movement is used in galloping manner.

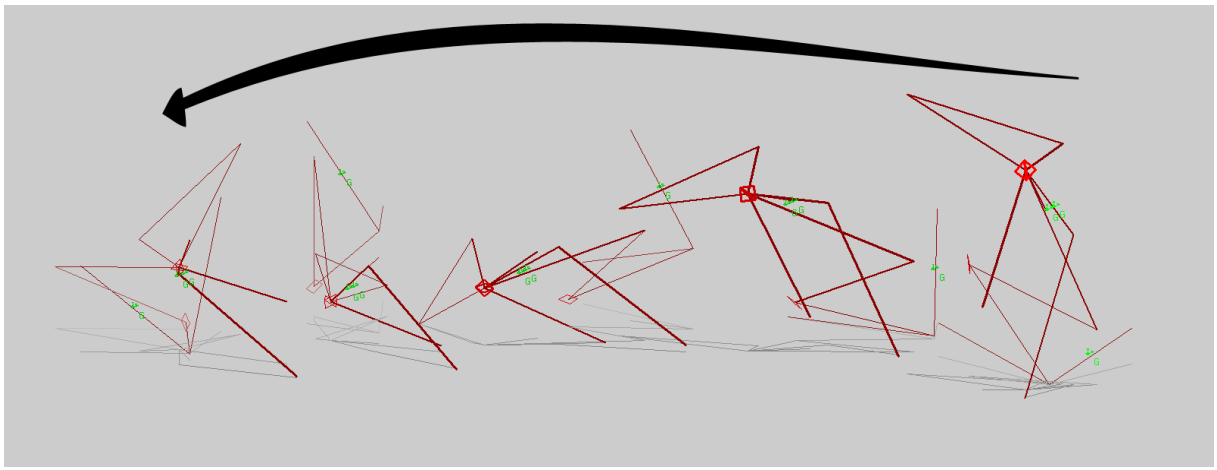


Figure 4.18. Somersault-like manner of movement. This creature combines rolling and periodical contraction and straightening to perform something similar to somersault.

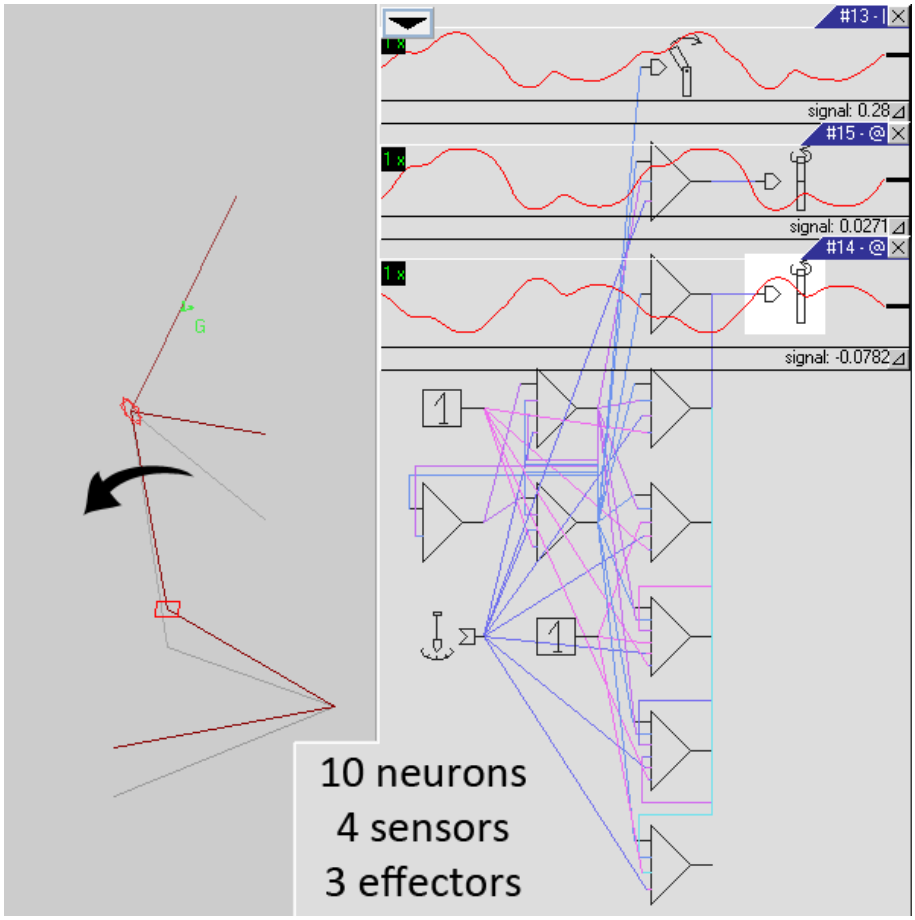


Figure 4.19. Example with high brain complexity yet simple body. Creature was created when alternating neuron number, we can see, that the signal directed to three effectors can be quite complex. This results in rolling movement of whole creature's body.

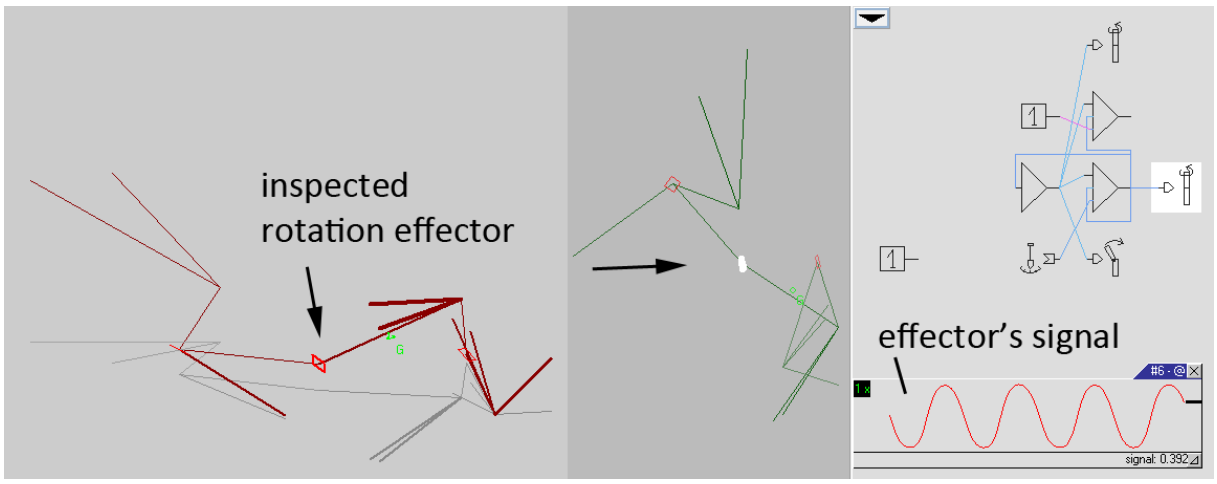


Figure 4.20. Inspection of creature's rotation effector. We can inspect input signal in each individual sensor, effector and neuron in creature's body. Here we can see sinus signal in rotation effector in part that connects creature's front and back segment of body. Creature hops on two legs.

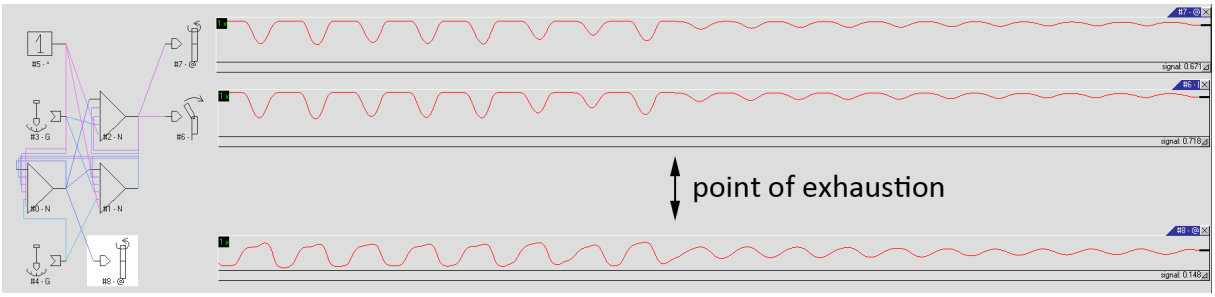


Figure 4.21. Problem of exhaustion of neuron signal in neural network after the regular lifetime period. Creature uses gyroscopic sensor in its own body to excite neural network's signals, which are then used by effectors. When certain threshold is achieved and the creature steps in way, that it loses its momentum, the gyroscopic signal ends and so does the neural network activity. This stumble occurs after the expected lifetime value, hence it wasn't solved during evolution. We can see it only when inspecting creature after experiment.

Chapter 5
Conclusion and future research

5.1 Conclusion

In this bachelor thesis I have followed the idea of Karl Sims work and successfully implemented evolutionary algorithm to develop virtual creatures. Unlike in his work I have used hybrid algorithm and different virtual creature structure. I have programmed Java application which uses the *Framsticks* simulator for creature evaluation. It uses *Framsticks* genotype language *f0* for communication with the simulator, however in our application we have developed and implemented abstraction layer, which helps us control the behavior of experiments. This abstraction layer guaranties production of valid virtual creatures and their conversion to genotype *f0*.

In our implementation of evolutionary algorithm we have created rules for mutation and crossover operations. I have studied the possible usage of hybrid algorithm and implemented altered version of evolutionary algorithm with the use of selection methods, fitness transformation and elitism. For individual optimalization we have used differential evolution algorithm compatible with chosen representation of neural networks. Possible two–phased structure of evolution which focuses on body and brain (neural network) evolution was suggested. We have declared referential values for parameters and proposed and carried out experiments to test the behavior of algorithm. Results of experiments were discussed and few interesting individuals were shown.

5.2 Suggestions for future work

For future research we suggest the possible exploration of alternative configuration of differential evolution. We could use different selection methods or population sizes to spare some computational requirements. We could also alter the structure of used hybrid evolution algorithm, or try and compare different sampling methods than roulette wheel selection.

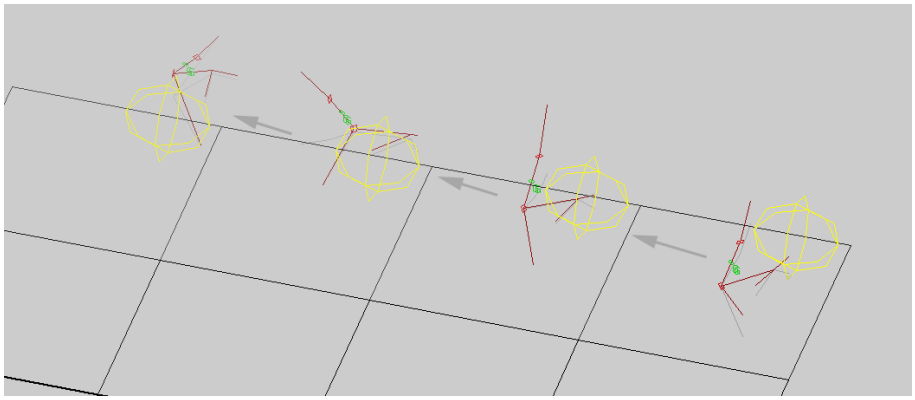


Figure 5.1. Example of food location in simulated world for guidance of virtual creatures.

In sense of experiment suggestions we propose the inclusion of food ingestion into world simulation. We could build it upon our second experiment (section 4.3) with walking individuals. We would need to allow appearance of smell receptors as usable sensors in mutation operations. These would lead creatures towards food and allow them to ingest food to refill their energy and live longer. Food ingestion is dependent only on creature's proximity to food. For fitness evaluation we could measure amount of food ingested or award the ability to follow food trail. Simple approach would be to depend on longer lifetimes of creatures walking in the direction of food and ingesting it, which would reflect in longer walked distances. The support (food, sensors, etc.) for such experiment is already included in the *FramScript* simulator, but the implementation of more complex fitness evaluation method would require some scripting done in *FramScript*. For experiment definition we propose situation similar with figure 5.1, where we placed food in predefined locations. This experiment would teach creatures to either walk in straight line towards food or learn to steer and follow path made from food.

We could implement learning process consisting from several worlds, where we would teach virtual creatures more complex behavior in gradual task definitions. We could start with walking and then use evolved individuals in several next worlds with the final goal of traversal through unknown worlds.

References

[1] Karl Sims. Evolving virtual creatures. In Dino Schweitzer, Andrew Glassner, and Mike Keeler, editors, *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 15–22, New York, NY, USA, 1994. ACM.
http://www.karlsims.com/papers/siggraph94.pdf.

[2] Karl Sims. Evolving 3D morphology and behaviour by competition. In R. Brooks and P. Maes, editors, *Artificial Life IV Proceedings*, pages 28–39, MIT, Cambridge, MA, USA, 6-8 July 1994. MIT Press.
http://karlsims.com/papers/alife94.pdf.

[3] J.B. Pollack and H. Lipson. The golem project: evolving hardware bodies and brains. In Jason Lohn, Adrian Stoica, Didier Keymeulen, and Silvano Colombano, editors, *Evolvable Hardware, 2000. Proceedings of the Second NASA / DoD Workshop on Evolvable Hardware*, pages 37–42, Palo Alto, CA, USA, 2000. IEEE Computer Society.
http://creativemachines.cornell.edu/papers/EH00_Pollack.pdf.

[4] Yoon-Sik Shim and Chang-Hun Kim. Evolving physically simulated flying creatures for efficient cruising. *Artificial Life*, 12(4):561–591, October 2006.
http://www.cc.gatech.edu/~turk/bio_sim/articles/evolve_flying_creatures.pdf.

[5] Thomas Geijtenbeek, Michiel van de Panne, and A. Frank van der Stappen. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics*, 32(6), 2013.
http://www.staff.science.uu.nl/~geijt101/papers/SA2013/SA2013.pdf.

[6] N. Lassabe, H. Luga, and Y. Duthen. A new step for artificial creatures. In *Artificial Life, 2007. ALIFE '07. IEEE Symposium on*, pages 243–250, April 2007.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.149.4345&rep=rep1&type=pdf.

[7] Dan Lessin, Don Fussell, and Risto Miikkulainen. Open-ended behavioral complexity for evolved virtual creatures. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2013*, 2013.
http://nn.cs.utexas.edu/downloads/papers/lessin.gecco13.pdf.

[8] Kenneth A. De Jong. *Evolutionary Computation: A Unified Approach*. Bradford Book. MIT Press, 2006. ISBN: 978-0-262-04194-2.

[9] Agoston E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2003. ISBN: 978-3-540-40184-1.

[10] Zdeněk Buk. *Continual Evolution Algorithm*. PhD thesis, Czech Technical University in Prague, 2012.
http://neuron.felk.cvut.cz/~bukz1/cea/buk-2012-dis.pdf.

[11] Vladimír Mařík, Olga Štěpánková, Jiří Lažanský, Ivan Zelinka, and collective. *Umělá inteligence (4)*. Academia, Praha, 2003, in Czech. ISBN: 80-200-1044-0.

[12] Jarmo Ilonen, Joni-Kristian Kamarainen, and Jouni Lampinen. Differential evolution training algorithm for feed-forward neural networks. *Neural Processing Letters*, 17(1):93–105, 2003.

[13] Ivan Zelinka. *Evoluční výpočetní techniky: principy a aplikace*. BEN - technická literatura, Praha, 2009, in Czech. ISBN: 978-80-7300-218-3.

[14] K.V. Price, R.N. Storn, and J.A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Natural Computing Series. Springer, 2005. ISBN: 978-3-540-20950-8.

[15] Maciej Komosinski and Szymon Ulatowski. Framsticks web site, May 2014. `http://www.framsticks.com/`.

[16] Maciej Komosinski. The Framsticks system: versatile simulator of 3D agents and their evolution. *Kybernetes: The International Journal of Systems & Cybernetics*, 32(1/2):156–173, 2003. `http://www.framsticks.com/files/common/Komosinski_FramsticksSystem_Kybernetes2003.pdf`.

Appendix A
CD Content

A.1 CD Content

Following folders are placed on CD accompanying this thesis.

- **/Java source project**
  Folder with NetBeans project of `VirtualCreaturesEvolution` program, which contains implementation described in chapter 3.

- **/Javadoc**
  Contains generated javadoc with class diagram.

- **/Ready-made distributions**
  Contains ready-made bundles for each experiment for easy repetion of experiments. Java application is runnable with comand line:

```
java -jar VirtualCreaturesEvolution.jar
<experiment-name> <path-to-framsticks-folder>
```

  Java application can be controlled by editing the `experimentConfig.txt` file (necessary syntax is explained in readme files for each experiment). Both Linux and Windows versions are provided.

- **/Thesis TEXsource**
  Folder contains sources for this thesis. Can be compiled with `pdfcsplain` of attached TexLive bundle.

- **/Experiment results**
  Contains all individuals from performed experiments (100 per each configuration).

- **/Matlab scripts**
  Attached Matlab scripts used for graph generation.

- **/Video presentation**
  Contains video presentation file of sample individuals. The same video can be viewed on `http://www.youtube.com/watch?v=qRhW1NPRn0g`.

- **/References**
  Folder containing all publicly accessible papers in this thesis. We chose to provide them in this folder for convenience purposes and for the case of their disappearance from provided link.

A.2 Software

These applications are available on corresponding webpages for free download, but the exact versions used while working on this thesis are placed into `/Software` folder.

- **Framsticks** current official release 3.2
- **Java** v1.7
- **NetBeans IDE** in version 7.0.1.
- **TeXLive** and **TeXworks** for opening TEXsources of this thesis.

Appendix B
Additional graphical content

Experiments often yielded multiple results and not all of them were significant for our explanations. Also sometimes several graphs shown the same principle or behavior and only the exemplary one was used for demonstration. Here we provide the rest of these results.

B.1 Experiment 2 – the fastest creature – number of neurons

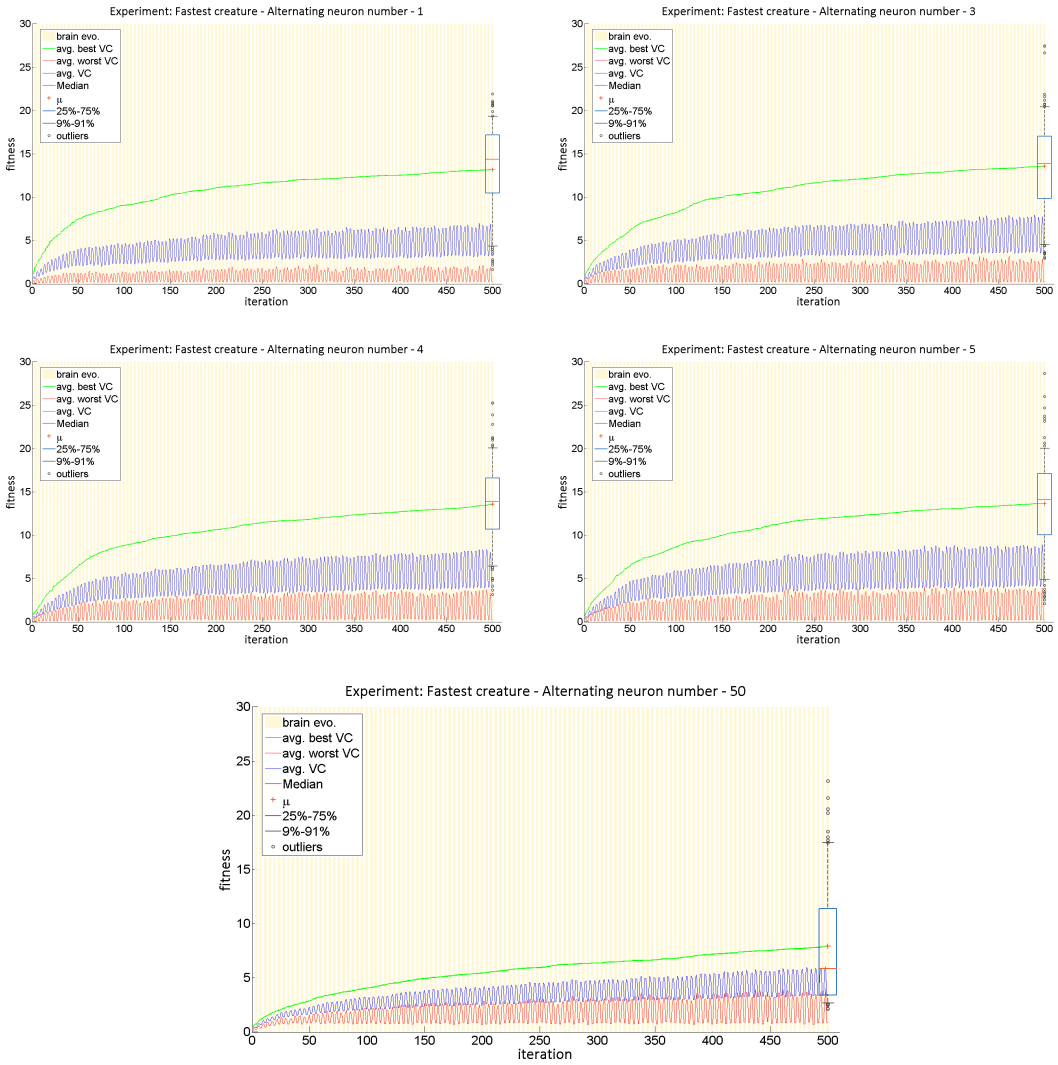


Figure B.1. Experiment 2 graph with alterating number of neurons. Graph comparing neuron counts of 2 and 10 is already on figure 4.7.

B.2 Experiment 2 – the fastest creature – iteration ratio

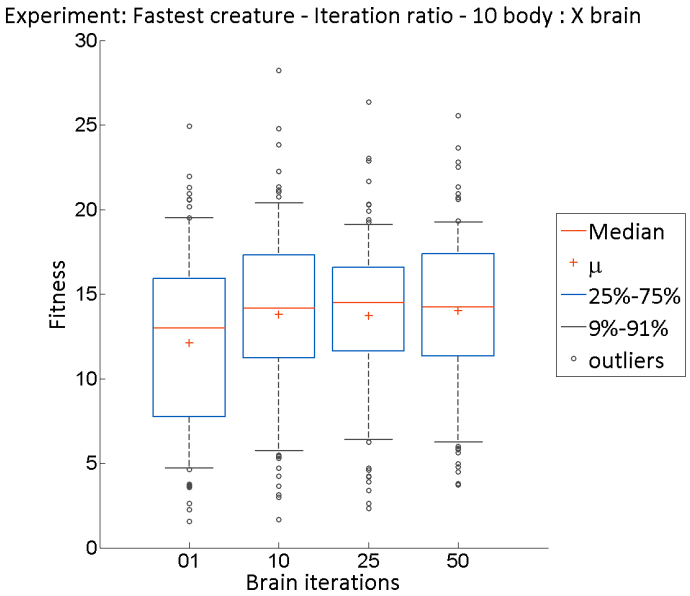


Figure B.2. Alternating brainIteration number with bodyIteration of 10. Similar behavior as 4.8.

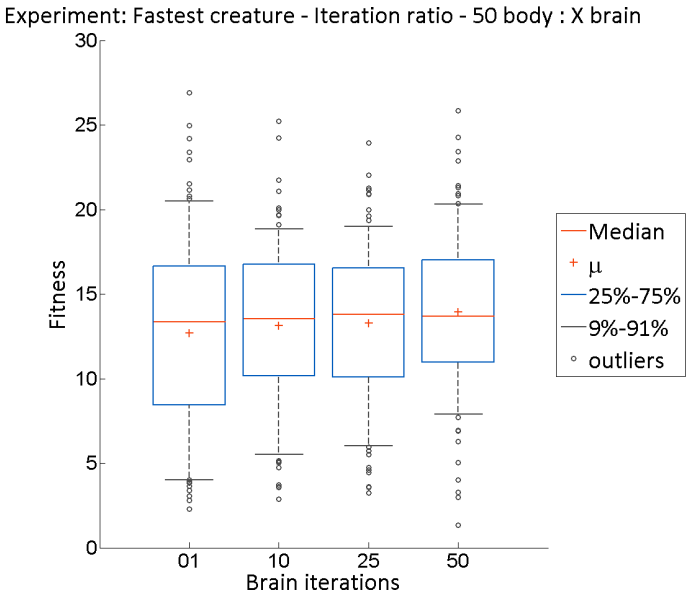


Figure B.3. Alternating brainIteration number with bodyIteration of 50. Similar behavior as 4.9.

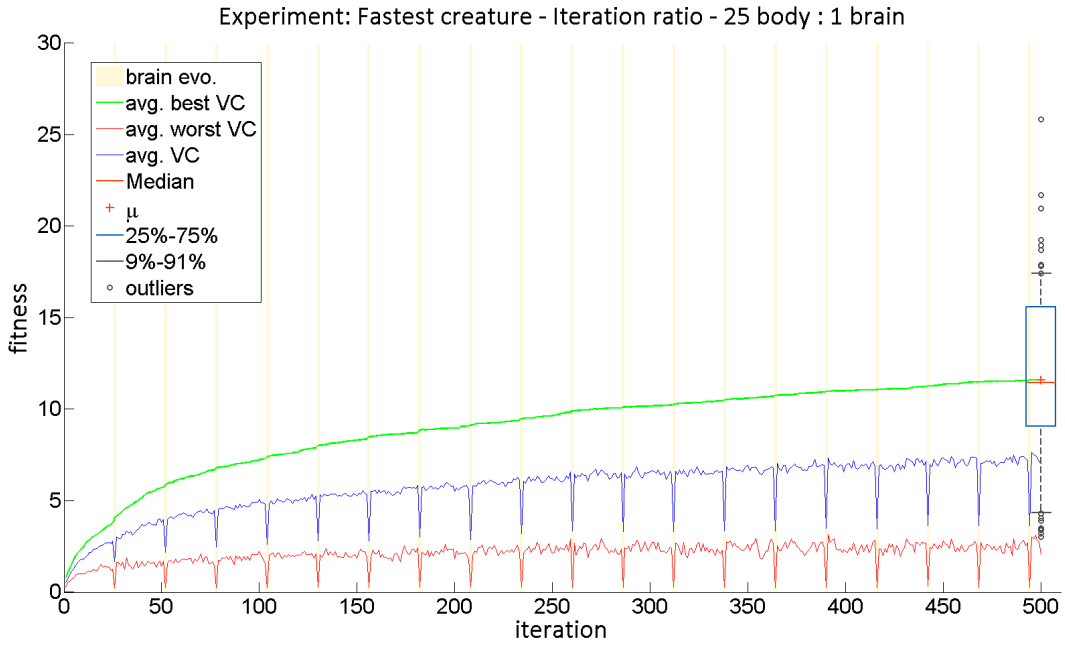


Figure B.4. Another unbalanced example – opposite extreme to 4.10. Compare with more balanced version in B.5.

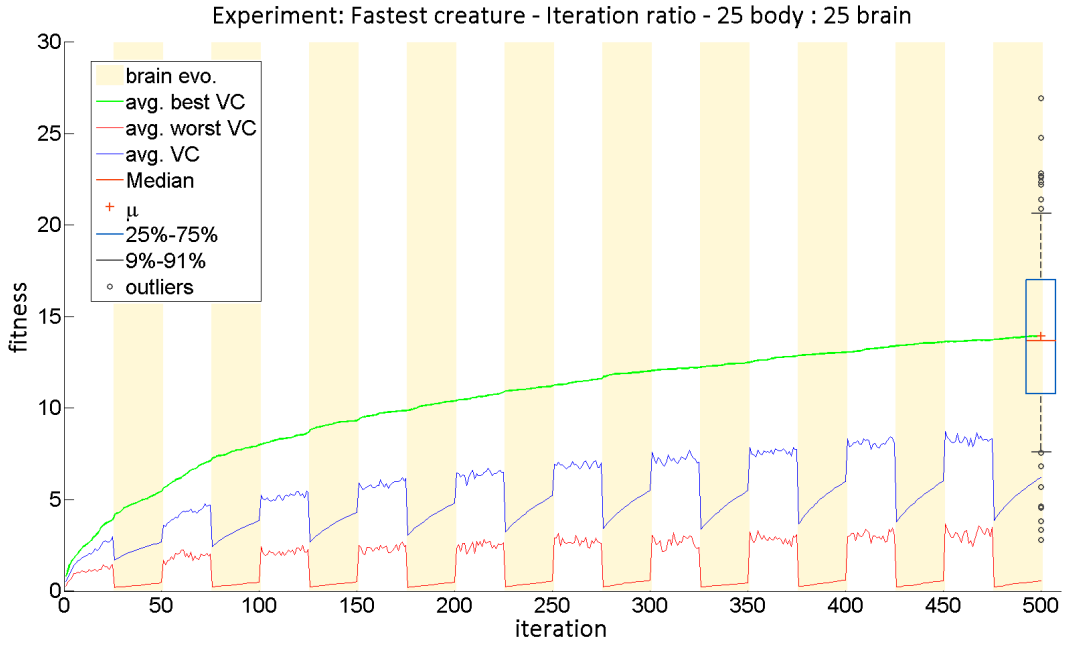


Figure B.5. Jump in efficiency when balance in body and brain iteration ratio is maintained. Balanced version as comparison for B.4.

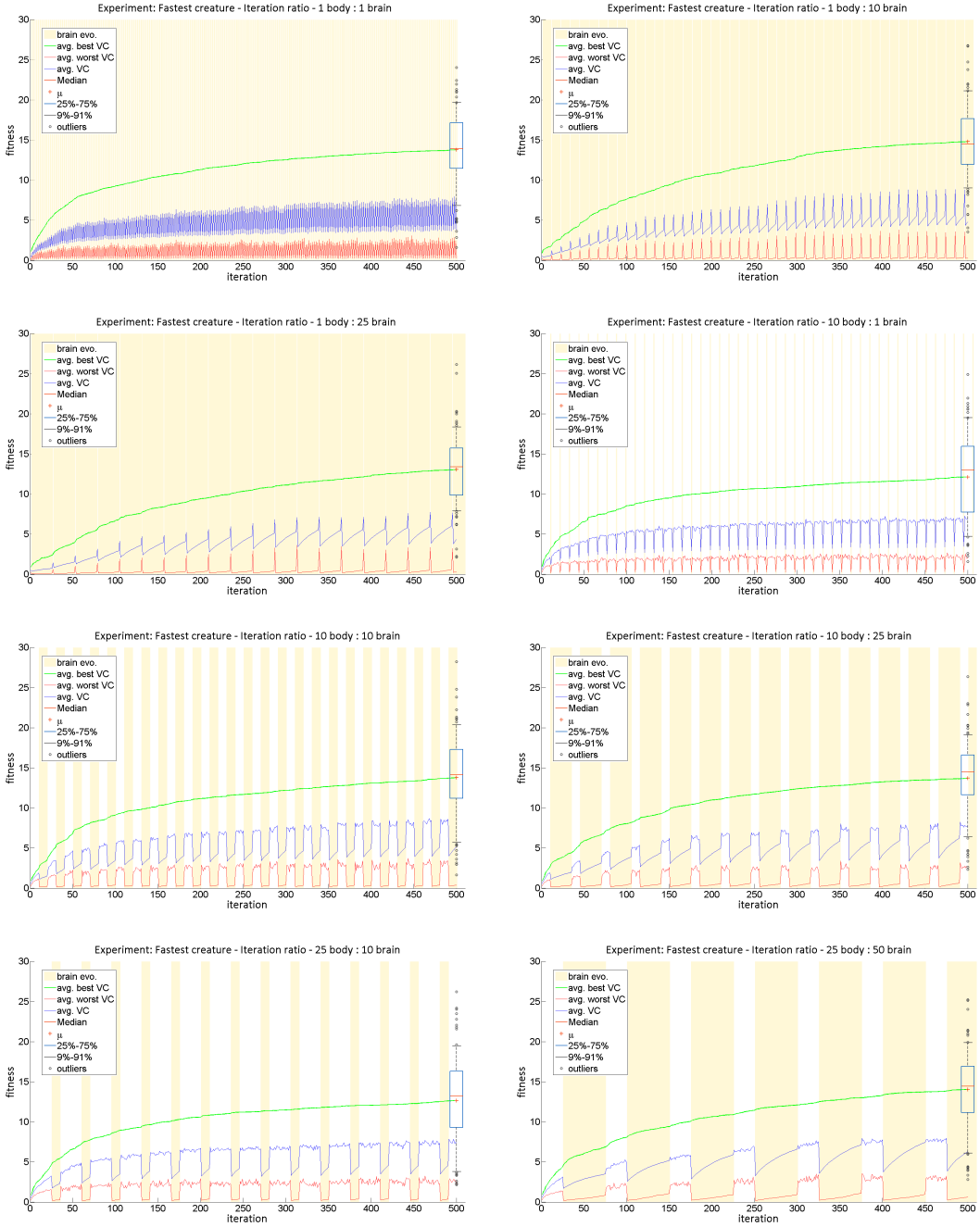


Figure B.6. Experiment 2 growth graph with alternating body : brain iteration ratio.

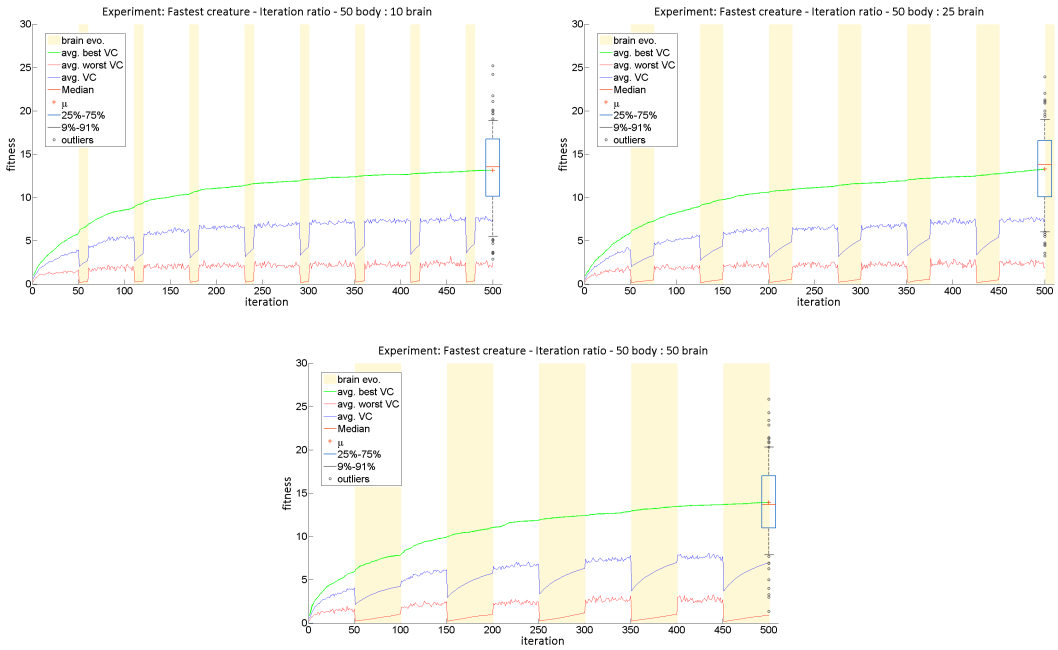


Figure B.7. Experiment 2 growth graph with alternating body : brain iteration ratio.

B.3 Experiment 2 – the fastest creature – popSize ratio

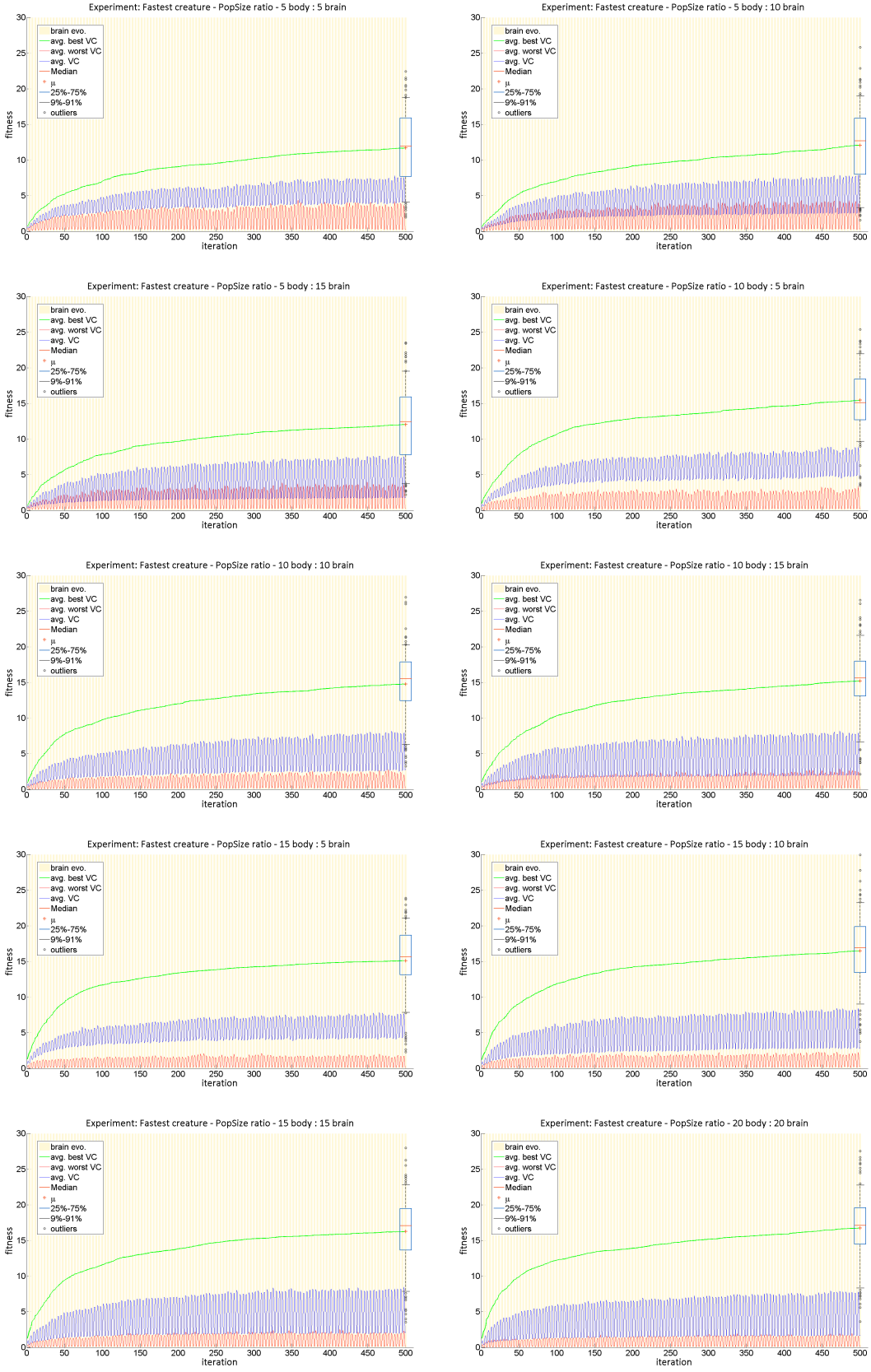


Figure B.8. Experiment 2 growth graph with alternating body : brain popSize ratio.