Computer Science Undergraduate Project
EPFL, Logic Systems Laboratory

June 21, 2003

Adam

A Modular Robot Evolution and Simulation Tool

Author: Daniel Marbach
Supervisor: Prof. Auke Ijspeert

Contents

1 INTRODUCTION 3

1.1 Modular Robots 3
1.2 Co-evolution 3
1.3 Robot Simulation 4

2 STATE-OF-THE-ART 4

2.1 Karl Sims block creatures 4
2.2 Modular Transformer 4
2.3 PolyBot 5
2.4 CONRO 6
2.5 Framsticks 6

3 INTRODUCTION TO ADAM 7

3.1 Modules 7
3.2 The Adam script 7
3.3 Structures 8
3.4 Parameters 11
3.5 Body parts 12
3.6 Formatting scripts 12
3.7 Example: A quadruped robot 12

4 GENETIC ALGORITHM 14

4.1 Phenotype space 14
4.2 Genetic Encoding 14
4.3 Genetic operators 15
4.4 Initialization 16
4.5 Selection and replacement 16
4.6 Algorithm 16

5 IMPLEMENTATION 17

5.1 Internal representation of robots 17
5.2 Control of powered hinges 17

6 RESULTS 18

7 CONCLUSIONS 19

1 Introduction

This chapter is a broad overview of the three main concepts of this work: Modular robotics, co-evolutionary algorithms and robot simulation. Chapter 2. focuses on the state-of-the-art in modular robot design and shows some current research projects.

1.1 Modular Robots

Modular robots (figure 1) are built by connecting modules together, much like a child's Lego bricks. Current research is done with robots built up from tens to hundreds of modules. In the future one can imagine robots with thousands, potentially millions of modules.

Modular reconfigurable robots have further the ability to change their structure by reconnecting the modules in different ways [3]. They offer many interesting qualities such as versatility, robustness and low cost.

They are versatile due to the many different ways modules can be connected together. Hundreds of modules allow usually millions of configurations, which can be applied to many diverse tasks. If the robot is self-reconfigurable, he can even adapt autonomously to different situations. Such a robot could for example move through a small hole in a 'snake configuration' and then change its structure to a four-legged creature to climb up a staircase.

Robustness comes from redundancy. Modular systems typically use a small number of module types but they are built of many modules. If some modules are damaged, overall function degrades but the system still works. Furthermore, self-repair mechanisms can be implemented, for example by adding spare modules that replace the faulty element or by reconfiguring the robot around it.

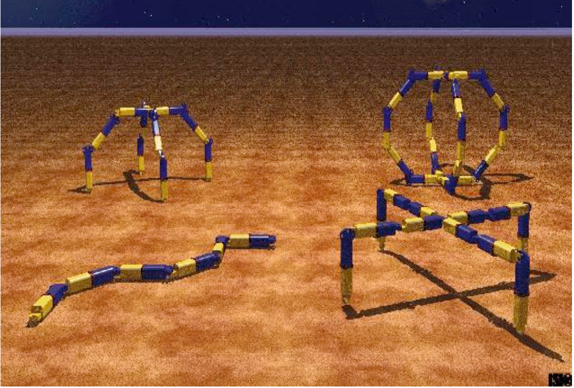


Figure 1: Some possible configurations of modular robots using simple hinge modules.

Potentially, the modules can be produced at a low price because they are small, relatively simple and used in large numbers. Another aspect is reuse: Suppose a robot is working on an assembly line. If the production process is changed, he can be reconfigured to suit his new task with the same modules.

All this promises to make modular reconfiguring robots very interesting for a wide range of tasks. Versatility and reliability make them especially well suited to harsh and unpredictable environments found in applications such as urban search and rescue operations after natural catastrophes or terrorist attacks, space exploration or battlefield reconnaissance.

1.2 Co-evolution

Traditionally, biologically inspired learning and evolutionary algorithms (EAs) only affect (if they're applied at all) the controller of the robot; the hardware is still totally designed by humans. Aspects of morphology like the shape, size, method of propulsion or type and position of sensors are all decisions the engineers make.

This unbalance in the use of EAs regarding morphology and control probably has practical reasons: Usually the controllers are designed for specific robots that already exist, engineering a completely new robot type is too expensive. On the other hand building and testing evolved hardware is much more complicated than doing the same for software. While it is possible to test an evolved controller immediately just by loading it onto the robot or even by doing the EA online (on the robot itself), one first has to build the evolved hardware to test it.

Nevertheless, if we have a look at nature, we see co-evolution on all levels. There's no such thing as a body and a brain god, each one doing his job one after the other. Evolution wouldn't suddenly come up with a leg and then slowly evolve the required sensors and controller for it. The powerful solutions nature came up with to allow creatures to survive in their complex environments were found with cooperative, co-evolutionary processes. There is co-evolution on several levels: The body is cooperatively co-evolving with the sensors and the brain and there's co-evolution between a species and its environment (for example predator-prey relationships).

There are many arguments if favor of body-brain co-evolution: Biologically inspired engineering techniques have proven to be effective and often result in better solutions than traditional approaches. If nature uses co-evolution, why shouldn't we do the same in robot engineering?

If the EA is applied only to the controller, it works on a static fitness landscape. Even if it finds the optima, the resulting robot is certainly not the best possible solution for his task. Co-evolution results in changing fitness landscapes over time and expands the search space to find solutions that are more powerful [4].

Modular robots are especially well suited to co-evolution. The complexity of designing the structure and programming the controller of the robot grows exponentially with the number of used modules. It might be impossible to configure and program a robot made up of thousands of modules by hand. Co-evolutionary algorithms allow us to evolve morphology and controllers simultaneously to suit a particular task.

1.3 Robot Simulation

Simulation plays an important role in evolutionary robotics. In general, it is impossible to calculate the fitness analytically. Therefore, the fitness has to be measured either on the real robot (online) or in simulation (offline).

The advantage of online EAs is the 'what-you-see-is-what-you-get' effect. Suppose we want to evolve a robot that is able to push a button. If a robot succeeds the EA could be stopped, naturally if the robot were tested now, it would succeed again provided the task is identical to the one in the EA. The problem with online EAs is that they are extremely slow.

Using a simulation to measure fitness values is much faster than testing the robots in the real world each time but they might have problems closing the 'reality gap'. A robot that perfectly accomplishes a task in simulation can be completely useless in the real world. To ensure a good transfer from simulation to real world the body of the robot and the environment must be carefully (not accurately) reproduced. There are a number of different approaches:

Adding noise from a uniform distribution centered about zero to the precise values produced by analytical models is the most simple and common way to facilitate transfer from simulation to real world. Unfortunately, the noise in the real world is not uniform and robots still might have problems closing the reality gap.

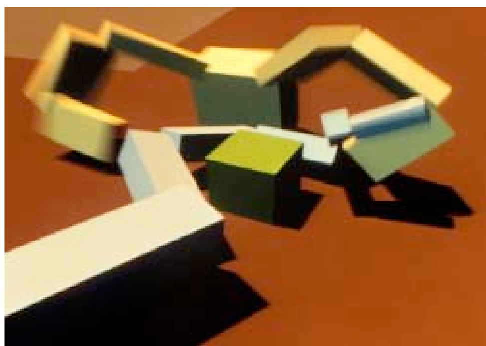


Figure 2: The famous block creatures of Karl Sims competing over control of the box in the middle.

Sampling guarantees a much better transfer. Values returned by the robot sensors for given objects and by actuators at given speeds are measured and stored in a look-up table. The simulation accesses these values and adds some noise.

Jakobi proposed another approach: Minimal simulations only try to model the base set features. These are the characteristics of the interaction between robot and environment that are relevant for the expected behavior. The remaining features are considered implementation-specific and therefore are simplified and varied randomly from one trial to the next so that evolution does not rely on them. Minimal simulations speed up significantly computing time and transfer well to the real world but require the programmer to know in advance, what the relevant features will be that must be accurately modeled [6].

2 State-of-the-art

2.1 Karl Sims block creatures

Karl Sims co-evolved body and brain of block creatures (figure 2) quite similar to our Adam robots already in 1994. The morphology and the neural systems for controlling their muscle forces are both genetically determined, and the morphology and behavior can adapt to each other as they evolve simultaneously.

The genotypes are structured as directed graphs of nodes and connections, and they can efficiently but flexibly describe instructions for the development of creatures' bodies and control systems with repeating or recursive components (figure 3). When simulated evolutions are performed with populations of competing creatures, interesting and diverse strategies and counter-strategies emerge [7] [4].

2.2 Modular Transformer

In 2002, the Distributed System Design Research Group of the National Institute of Advanced Industrial Science and Technology of Japan developed a self-reconfigurable modular robot (Modular Transformer). It successfully realized multi-mode robotic motion by changing its shape smoothly from a crawler to a four-legged walking robot (figure 4).

The used modules are very similar to the Adam hinge module. They consist of two semi-cylindrical parts that are connected by a hinge. Each semi-cylindrical part can rotate 180 degrees and has three connection surfaces, which can connect to other modules by magnetic force. In addition, electrodes are placed on each connection surface for power supply and communication with the host computer [8].

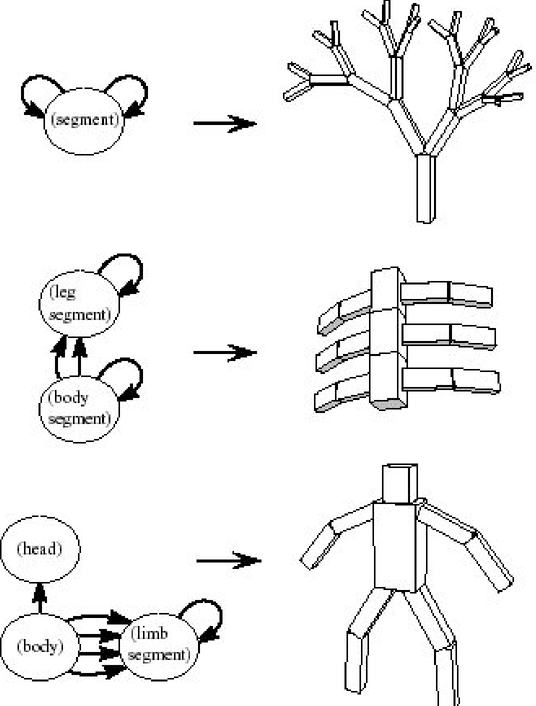


Figure 3: The genotype of Karl Sims block creatures is a directed graph.

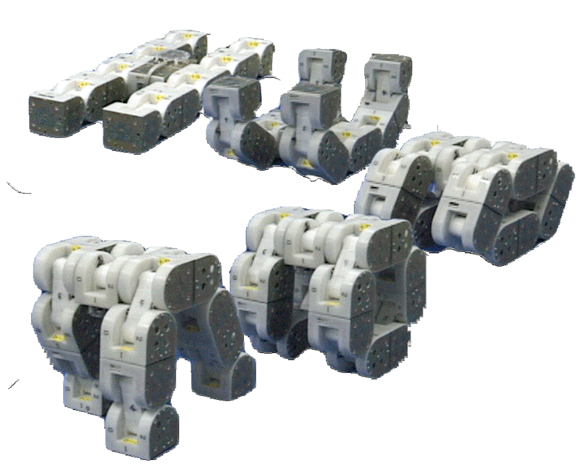


Figure 4: The Modular Transformer of the AIST is capable of changing its shape from a crawler to a four-legged walking robot.

2.3 PolyBot

The Palo Alto Research Center (PARC), a subsidiary of Xerox Corporation, developed PolyBot (figure 5). The modules of the first generation were manually screwed together, so they did not self-reconfigure. The G1 modules showed the first instance of simple reconfiguration for locomotion in 1997. The modules of the second generation have the ability to automatically attach and detach from each other making them self-reconfigurable. PolyBot is the first robot to demonstrate sequentially two topologically distinct locomotion modes by self-reconfiguration. The third generation of PolyBot is currently under construction. The goal is to demonstrate reconfiguration and locomotion with large configurations. PARC uses two module types: Hinges and connection modules. On both sides of the hinge, only one module can be attached. Connection modules are cubes; other modules can be fixed on all six faces.

PolyBot has shown versatility, by demonstrating locomotion over a variety of terrain and manipulating a variety of objects. It has successfully completed obstacle courses including ramps, steps, pipes and chicken wire [9].

Figure 5: The second generation of PolyBot is able to self-reconfigure itself to spider, crawler, loop and snake configurations.

2.4 CONRO

The CONRO Project of the USC Information Sciences Institute has a goal of providing the Warfighter with a miniature reconfigurable robot that can be used to perform reconnaissance and search and identification tasks in urban, seashore and other field environments. CONRO (figure 6) is homogenous (made from identical modules) that can be programmed to alter its topology in order to respond to environmental challenges such as obstacles.

The base topology is simply connected, as in a snake, but the system can reconfigure itself in order to grow a set of legs or other specialized appendages. Each module consists of a CPU, some memory, a battery, and a micro-motor plus a variety of other sensors and functionality, including vision and wireless connection and docking sensors. The control mechanism is distributed and hormone-based [10].

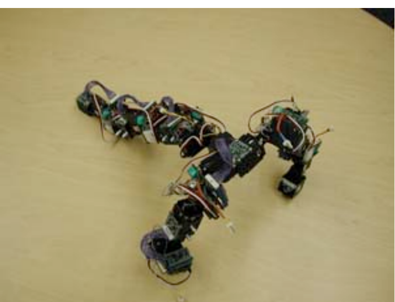


Figure 6: CONRO self-reconfiguring itself from a snake to a two armed crawler configuration.

2.5 Framsticks

Framsticks is a three-dimensional life simulation project. Both physical structures ("bodies") and control systems ("brains") of creatures are modeled. It is possible to design various kinds of experiments, including simple optimization (by evolutionary algorithms), co-evolution, open-ended and spontaneous evolution, distinct gene pools and populations, diverse genotype/phenotype mappings, and species/ecosystems modeling.

Framsticks creatures are built of sticks. Modificators can be applied to change the properties of the sticks. The genotype encodes all parameters of a creature, including morphology, sensors and actuators placement as well as control network. There are gyroscope, touch and smell sensors. Actuators can rotate or bend sticks.

Adam and Framsticks have very similar philosophies, the main difference being that Adam focuses on robotics while Framsticks focuses on artificial life. Both systems use simple modules to construct creatures and the genotype encodes both structure and control. Currently there's no notion of energy used in Adam. Framsticks creatures consume energy and die if they don't find a 'food' source. A creature can also kill another one and steal its energy [11].

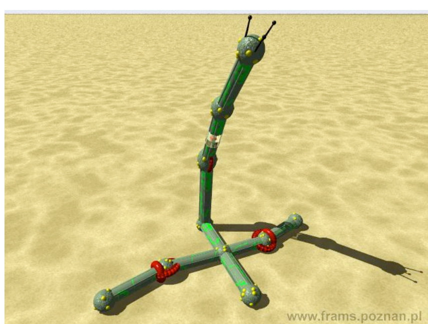


Figure 7: A Framsticks creature with a sense of touch (on the top) and a sense of equilibrium (glass-like cell).

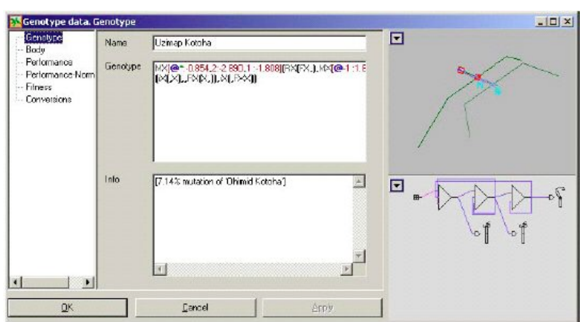


Figure 8: Framsticks user interface allows editing the genotype and watch the structure and control network change in real time.

3 Introduction to Adam

Adam is a system to evolve and simulate modular robots. Currently the robots are not reconfigurable and loop configurations like the one of PolyBot in figure 5 are not allowed. The simulation accurately models rigid body dynamics (kinematics, gravity, friction, collisions, etc) in a world that consists simply of an infinite plane. In future versions of Adam reconfiguration and cycles will be supported and the simulation might offer environments that are more complex.

3.1 Modules

Adam robots are built with a number of different, well-defined module types. Modules are like LEGO blocks: Each type has certain qualities and positions where it is possible to put another building stone. Adam is built to make it easy to add new types of modules. One could define modules that correspond to real hardware and then actually build the evolved robots but for now, we test the system with only a single element type.

The hinge module (figure 9) consists of two cubes, fixed together with a hinge. Other elements

Parameter	Definition
Initial angle	A hinge has an initial angle between −150.0 and 150.0 degrees. High and low stops as well as the oscillation of the desired angle are all relative to the initial angle.
Low stop	The maximal negative deviation of the hinge with respect to the initial angle. The low stop must be set between 0.0 and 150.0 degrees.
High stop	The maximal positive deviation of the hinge with respect to the initial angle. The high stop must be set between 0.0 and 150.0 degrees.
Powered	A Boolean indicating if the hinge is powered with a motor or rigid.
Maximal force	The maximal force of the motor (only relevant for powered hinges).
Amplitude Frequency Phase	Amplitude, frequency and phase of the oscillation of the desired angle (only relevant for powered hinges).
Elastic	A Boolean indicating if the hinge is elastic or not. Note that elasticity does not have any effect on powered hinges.
Spring	The spring constant of an elastic hinge.
Damping	The damping constant of an elastic hinge.

Table 1: Definition of hinge parameters

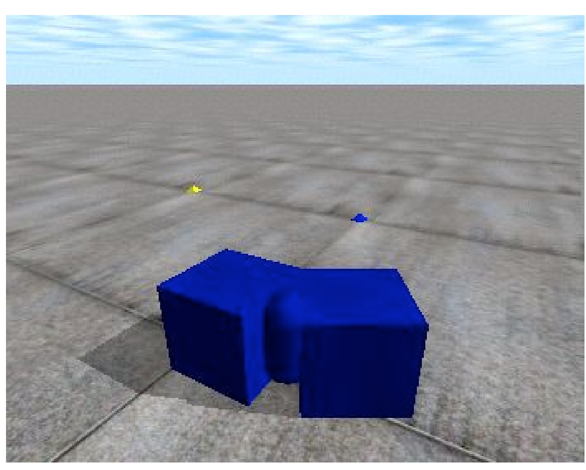


Figure 9: A rigid Adam hinge module with an initial angle of 30 degrees in the simulation world.

can be attached at each one of the ten free faces.
The hinge can be rigid or powered by a motor. Each powered hinge has a PD controller on its angle. Furthermore, the joint can be elastic or not.

The desired angle α of a powered hinge is defined with the amplitude A, frequency f and phase φ of a sinus oscillation:

$$\alpha = A \sin(2\pi f t + \varphi)$$

Naturally, if α gets outside the bounds set by the low and high stop, the hinge can't follow the desired trajectory and will wait at the stop until α reenters the interval. The spectrum of movements of the robots is enriched by the option of setting the low and high stops purposely smaller than the amplitude.

Unpowered hinges can be elastic. In this case, the joint applies a spring force defined by the spring and damping constants.
Refer to chapter 5 for more detailed information about the implementation of these features.

3.2 The Adam script

The user can define his own Adam robots with a simple script. Furthermore, this allows evolved robots to be saved in plain text format. They can then be inspected and edited by the user with the text editor of his choice.

As mentioned above, Adam robots are built with modules that can be fixed together like LEGO blocks . Each module type defines the characteristics of the element such as how it can be fixed with other modules. An Adam robot is valid if the same structure could be built in the real world with corresponding elements.

```
script        = {  ident '{' bodypart '}'  }
                 [headOrient] bodypart
headOrient    = 'ORIENTATION'
                 '(' real ',' real ',' real ',' real ')'
bodypart      = 'STRUCTURE' structure
                 'PARAMETERS' parameters

structure     = (   [pos] [orient] module
                     {  pos limb
                         | [pos] [orient] module } )
                 | [pos] [orient] ident
limb          = '(' structure ')'
module        = hinge
hinge         = 'H' 'identifier'
orient        = 'N' | 'E' |  'S' |  'W'
pos           = 'P0' | 'P1' | 'P2' | 'P3' | 'P4'
                 | 'P5' | 'P6' | 'P7' | 'P8' | 'P9'

parameters    = {  module function  }
function      = '.' 'identifier' '(' real arguments ')'
arguments     = [ ',' 'real' ]
```

real is a real valued number with the usual syntax (scientific notation is supported). There's one restriction: The number shouldn't start with a point, so write '0.4' instead of '.4'

identifier is a string with lower or uppercase letters. Underscores are allowed as well.

Quick reminder of grammars and EBNF:

A syntactic (non-contextual) grammar is defined by a set of terminal and non-terminal symbols, a set of productions and an initial symbol. It defines a language as the set of finite sequences of terminal symbols that can be derived from the initial symbol by successively applying productions. Repetitions can be expressed with {x} (zero or multiple occurrences of x), options with [x] (zero or one occurrence of x).

Figure 10: Syntactic grammar in EBNF

Figure 8 shows the syntactic grammar of the Adam script in Extended Backus Naur Form (EBNF). Don't worry if you are not familiar with EBNF, just read the next two sections and study table 2, which shows basic structures with their corresponding script expressions. If you have Adam installed you can build and simulate robots defined with scripts by typing: *Adam –open file*

The first part of the Adam script defines the structure of the robot (how the modules are connected to each other). Each module in the structure part must be given a unique identifier. The first letter of the identifier indicates which type it is. Hinge identifiers must always begin with a capital 'H'. These identifiers are used in the second part of the script to set the parameters of the modules.

3.3 Structures

In this section I will explain how Adam structures can be defined using the script. As mentioned above, scripts must have a structure as well as a parameter section. The examples in this section represent only the *structure* expression as defined in the grammar. If you want to build them with Adam, begin your file with the keyword 'STRUCTURE' followed by such an expression. The *structure* expression must be followed by the keyword 'PARAMETERS'. As you can see in the grammar, the section following this keyword can be left empty. In that case, all the modules will have default parameter values.

The most natural way for humans to write or read a description of a configuration consisting of many elements is a sequential building plan. Such a building plan is usually structured as a series of pictures showing which elements have to be added at which position to the part that has already been built. The Adam script follows this philosophy with the only difference that we can't use pictures and that always only one new module is added to the structure at a time.

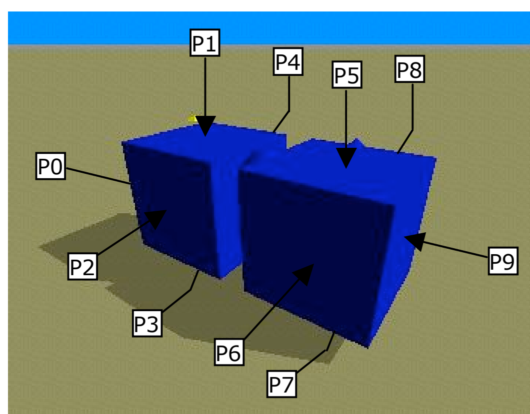


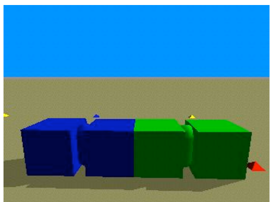
Figure 11: The hinge module has 10 positions where another module can be attached. Each position can be identified with respect to the local coordinate system of the hinge by its number as indicated in the figure.

P0: First cube back face
P1: First cube top face
P2: First cube right face
P3: First cube bottom face
P4: First cube left face
P5: Second cube top face
P6: Second cube right face
P7: Second cube bottom face
P8: Second cube left face
P9: Second cube front face

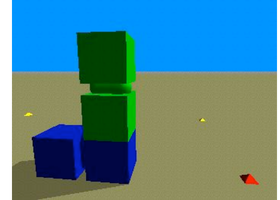
The key to understanding *structure* expressions and the rest of this section is the difference between 'attached AT', 'attached TO' and 'attached WITH'. Suppose that in our building plan, a new hinge Hb has to be attached to another hinge Ha that is already part of the unfinished structure. We say that Hb is attached TO Ha WITH a certain position and orientation. With respect to Ha, we could say that Hb was attached AT a certain position. The first element of the *body* expression is the head of the robot. It is the only module that is not attached TO another one. Naturally, other elements can be attached AT various positions. All other elements are attached TO one and only one other module WITH a specific position.

I will now explain how a new hinge Hb can be attached to another hinge Ha. In the following examples, Ha is always the head of the robot. Naturally, the same syntax is used when attaching a hinge to an arbitrary module of a partially built robot. Build the examples with Adam if you have the possibility. In table 2 you can find a more exhaustive list of simple expressions and their corresponding phenotype. In this section, colors have no specific meaning. They are only used to distinguish the modules better.

The simplest way to attach the two hinges together is in a straight row. *'Ha Hb'* tells Adam to attach Hb to Ha using default positions. Hb can be attached at any free position of Ha. This is done by specifying the position and putting the new module in parenthesis. *'Ha P5 (Hb)'* attaches Hb at the top face of the second cube (position 5).

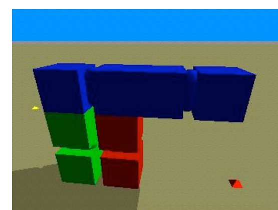


Ha Hb

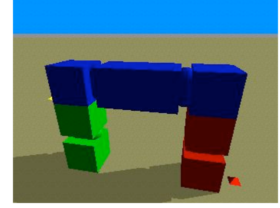


Ha P5 (Hb)

The part within the parenthesis is called a limb of Ha. In the previous example, the limb consists of only one module but in general, it can be any legal *body* expression. You can attach as many limbs at a hinge as there are free positions. The example on the left attaches two limbs at the downside of the first and second cube of Ha (positions 3 and 7) and a third hinge at the default position. On the right, the second limb is attached at Hd instead of Ha.

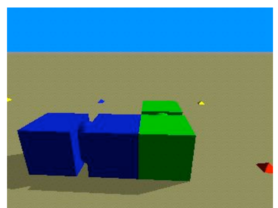


Ha P3(Hb) P7(Hc) Hd

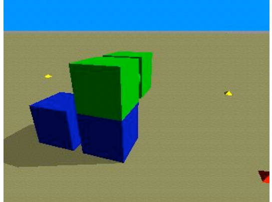


Ha P3(Hb) Hd P7(Hc)

Now that we have seen how to attach limbs at various positions of a module, let's see how a hinge can be attached with other positions than the default one. *'Ha P4 Hb'* attaches Hb with the left side of the first cube (position 4) to Ha. Naturally, it is still possible to specify where Hb should be attached to Ha. The example below on the right attaches Hb with position 4 to position 5 of Ha. Note that the position before the parenthesis is a face of Ha while the position within the parenthesis is one of Hb.

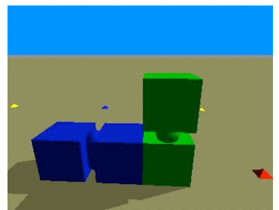


Ha P4 Hb

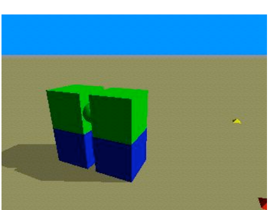


Ha P5 (P4 Hb)

Finally, once the faces that will be fixed together are specified, there's the possibility to set four different orientations for the second hinge (N, E, S and W for North, East, West and South respectively). Let's modify the previous examples to change the orientation of Hb from default to East. The hinges are still attached together with the same positions, but Hb has been rotated 90 degrees around an axis perpendicular to the faces of contact.



Ha P4 E Hb



Ha P5 (P4 E Hb)

I mentioned default positions several times already. Every Adam module type has to define a default position to attach limbs as well as a default position and orientation with which it gets attached to other modules. This makes the scripts shorter and easier to read.

For hinges, these default values are the front face of the second cube (position 9) for limbs to be attached at and the back face of the first cube (position 0) with orientation 'north' to attach it to another element. This allows us to put two hinges in a straight row, the most natural and often used configuration, with the simple expression *'Ha Hb'*. The same structure can also be built by explicitly naming the positions and orientation:

Ha Hb
= Ha P9 (Hb)
= Ha P0 Hb
= Ha N Hb
= Ha P9 (P0 N Hb)

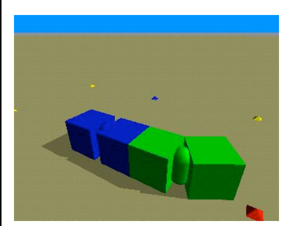
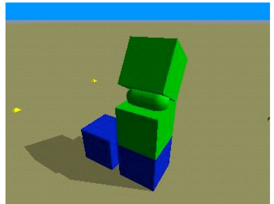
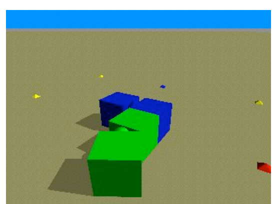
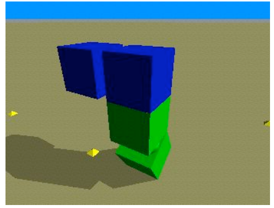
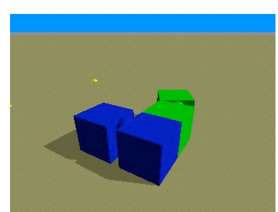
	*Ha Hb = Ha P9(Hb)* Attach Hb at the default position, which is the front face of the second cube of Ha.
	*Ha P5(Hb)* Attach Hb at the top face of the second cube of Ha.
	*Ha P6(Hb)* Attach Hb at the right side of the second cube of Ha.
	*Ha P7(Hb)* Attach Hb at the bottom face of the second cube of Ha. This picture is the first frame of the simulation, naturally this structure will fall over immediately.
	*Ha P8(Hb)* Attach Hb at the left side of the second cube of Ha.

Table 2: The five ways to attach a hinge AT the second cube of another hinge. Use positions zero to four to attach it to the first cube.

Hb has been given an initial angle of 30 degrees to show its orientation better. The various positions are obtained by rotating Hb in the default position (first example) plus or minus 90 degrees around the x-, y- or z-axis.

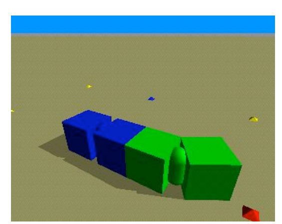
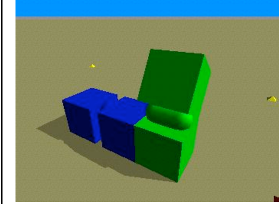
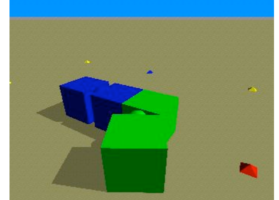
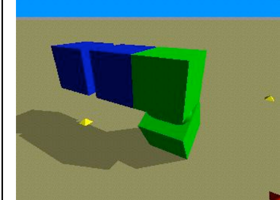
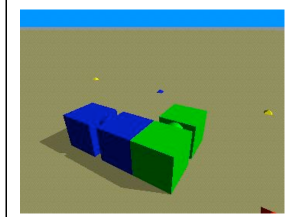
	*Ha Hb = Ha P0 Hb* Attach Hb with the default position, which is the back face of its first cube.
	*Ha P1 Hb* Attach Hb with the top face of its first cube.
	*Ha P2 Hb* Attach Hb with the right side of its first cube.
	*Ha P3 Hb* Attach Hb with the bottom face of its first cube. Naturally this structure falls over immediately after simulation starts.
	*Ha P4 Hb* Attach Hb with the left side of its first cube. If the initial angle of Hb is set to 30 degrees, this structure is invalid because of self-collision (See chapter X).

Table 3: The five ways to attach Hb WITH different positions of its first cube to Ha. Use positions five to nine to attach Hb with its second cube to Ha.

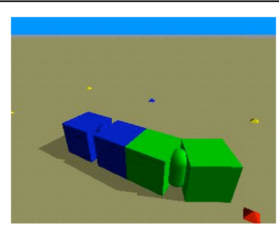
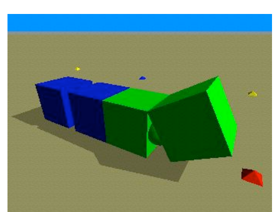
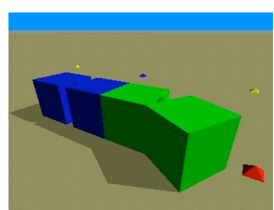
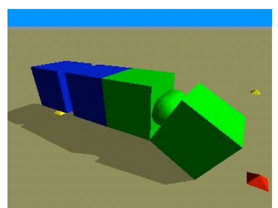
	*Ha Hb = Ha N Hb* Attach Hb with the default orientation North. The z-axis of Hb is facing up.
	*Ha E Hb* Attach Hb with orientation East. The z-axis of Hb is now directed towards the viewer.
	*Ha S Hb* Attach Hb with orientation South. The z-axis of Hb is now facing down.
	*Ha W Hb* Attach Hb with orientation West. The z-axis of Hb is now directed away from the viewer.

Table 4: Two hinges can be fixed together with two specific faces in four ways. Orientations are labeled North, East, South and West. They correspond to rotations of the default orientation North by 90, 180 and 270 degrees respectively around the x-axis of the second hinge.

3.4 Parameters

Now that we have seen how the structure of a modular robot can be defined, let's see how the parameters of the modules can be set. Remember that each module has been given a unique identifier in the *structure* part. These identifiers can now be used to set the parameters of the module with the following notation:

identifier.function(arguments)

All hinge parameter-setting functions currently available are defined in Table 5. The arguments are real numbers separated with commas. You can use scientific notation, but you shouldn't start the numbers with a point (write '0.5' instead of '.5')

Parameters of various modules can be set in any order. You can, for example, group all setters for a specific module together or order them by function type. The parameter setting functions can be separated by spaces, but I recommend using a new line for each one.

For internal reasons (see chapter X) it is not possible for a powered hinge to be elastic. If a hinge is set to be powered and elastic at the same time, the elasticity and damping constant will be ignored and the hinge behaves exactly the same way as if it were just powered and not elastic.

It is only necessary to set parameters that are different from the default value. Hinges with default parameters are rigid and have an initial angle of zero degrees. It is also possible to redefine the default parameter values using the same syntax as for setting parameters of a specific module. Instead of the identifier of a module, the indicator of the module type is used. For hinges, this is a capital 'H'. For example, you could set all hinges except Ha to be elastic with a spring constant of 10 and a damping constant of 1 like this:

PARAMETERS
H.soft(true, 10, 1)
Ha.soft(false, 0, 0)

As mentioned above, the values of the elasticity and damping constant of a rigid hinge do not have any effect. In the previous example they were set to 0, we might as well have chosen 1 or any other value.

initAngle(α) Sets the initial angle of the hinge to α.
powered(isPowered, loStop, hiStop, Fmax, A, f, φ) Sets all parameters that are relevant for powered hinges. *isPowered* indicates if the hinge has a motor or not. If this argument is equal *false*, the values of the other arguments have no influence on the behavior of the hinge. *LoStop* and *hiStop* are the low and high stops of the hinge, *Fmax* is the maximal force of the motor and *A, f* and φ are the amplitude, frequency and phase of the desired oscillation.
soft(isSoft, elast, damp) Sets all parameters that are relevant for elastic hinges. *isElastic* determines if the hinge will be rigid or elastic, *elast* and *damp* are the elasticity and the damping constant of the hinge.

Table 5: The three parameter-setting functions of the hinge module. Refer to chapter 3.1 for exact definitions of all hinge parameters.

However, currently it is mandatory to specify all arguments. Opening a script with the following parameter section results in an error:

PARAMETERS
H.soft(true, 10, 1)
Ha.soft(false)   // ILLEGAL

The parser prints out error messages if the syntax of the script is not correct. Usually the error message indicates what was expected and what was found. Naturally, this holds for the whole script and not just for the parameter section. If you open a file with the previous illegal statement, the following error message will be displayed in the terminal:

Expected ',' found ')'

Currently the parser does not count lines and words and can't indicate exactly where the error occurred. If you don't find it, try again with the *–debug* option. Now you can see how far the parser got and where the error is:

PARAMETERS
--> parameters
H.soft(true, 10, 1)
Ha.soft(true
--> Expected ',' found ')'

3.5 Body parts

The Adam script offers the possibility to define body parts that can be reused several times in the main structure. For example, you can define a leg or an arm at the beginning of the script and then attach these predefined structures at different positions with different orientations to the robot. The syntax is the same as for the main bodypart. To declare a bodypart, start with its identifier, open a left brace followed by the structure and parameter section as described in the previous chapters and finish with a right brace:

ident {
  STRUCTURE
  …
  PARAMETERS
  …
}

The bodypart can now be attached with various positions and orientations using the same syntax as if it were a simple module. Remember that what you're actually doing is attaching the head of the bodypart, which is a hinge. Default values can therefore be used as described in the previous chapters:

// attach with default position and orientation
Ha P1(ident)
// attach with position 3 and orientation East
Ha P1(P3 E ident)

3.6 Formatting scripts

As in every script or programming language, it is possible with the Adam script to write the same thing in many different ways. Some of them are easy to read, others not.

The Adam parser ignores white spaces, tabs and new lines; you are therefore free to format your scripts as you want. In this chapter, I propose one good way to do it. This is also, how the examples in the following chapters are formatted and how Adam saves evolved robots to text files. Formatting is crucial for the structure part, it's not so important for the rest of the script. The parameter section is formatted, quite naturally by beginning a new line for every parameter setting function. Formatting of the structural part can be done with the following three rules:

1. *Begin a new line for every limb.*
2. *Every time you begin a new line, indent by i. At the beginning, i is zero.*
3. *Increase i by one for the limbs of a module.*

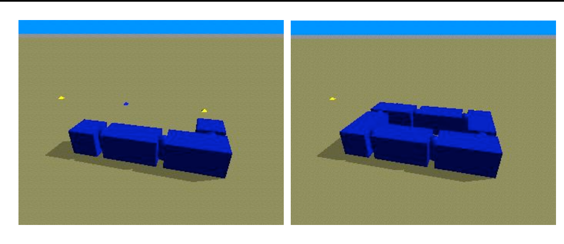
Remember that only structures in parenthesis are considered limbs. Just go on reading the following chapters if this seems somewhatcryptic and you will get used to writing nice scripts even without knowing these rules by heart. The advantage of writing structure expressions like this is that one immediately sees where a limb begins, ends and where it is fixed. Furthermore, the tree structure of the robot (see chapter 4.2) is quite visible.

3.7 Example: A quadruped robot

We will now apply the learned concepts to build a walking quadruped robot. Unfortunately, Adam can't yet show the robot growing as the script is written like Framsticks. Therefore, we always have to write a part of the script, save it to a file and then build it with Adam to check if the structure is correct. Let's begin with the body of the robot, a rectangle of rigid hinges (figure 12). Since Adam doesn't yet support cycles, the first and the last hinge can't be fixed together.

Let's now attach a leg at the right face of the first cube of H_body0 (figure 13). Three more legs could be added in the same way but it is much nicer to define the leg as a modular bodypart and then attach it four times at the desired positions (figure 14).

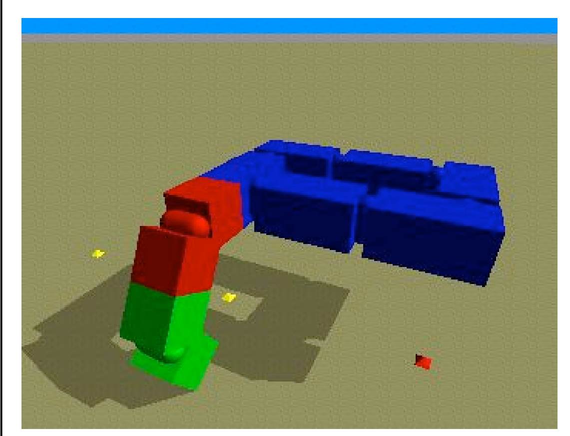
If you simulate this robot, you will see that it does push-ups and doesn't walk because all four powered hinges of the legs have the same oscillation. In future versions of Adam it will be possible to seed populations and to evolve only certain parameters of the robots while leaving the others untouched (see chapter X). We could now seed a population with this robot and hope that evolution will somehow take advantage of it, or we could evolve only the phases of the powered hinges to get this specific structure to walk.



```
STRUCTURE
Hbody0 Hbody1
P4 Hbody2
PARAMETERS
```

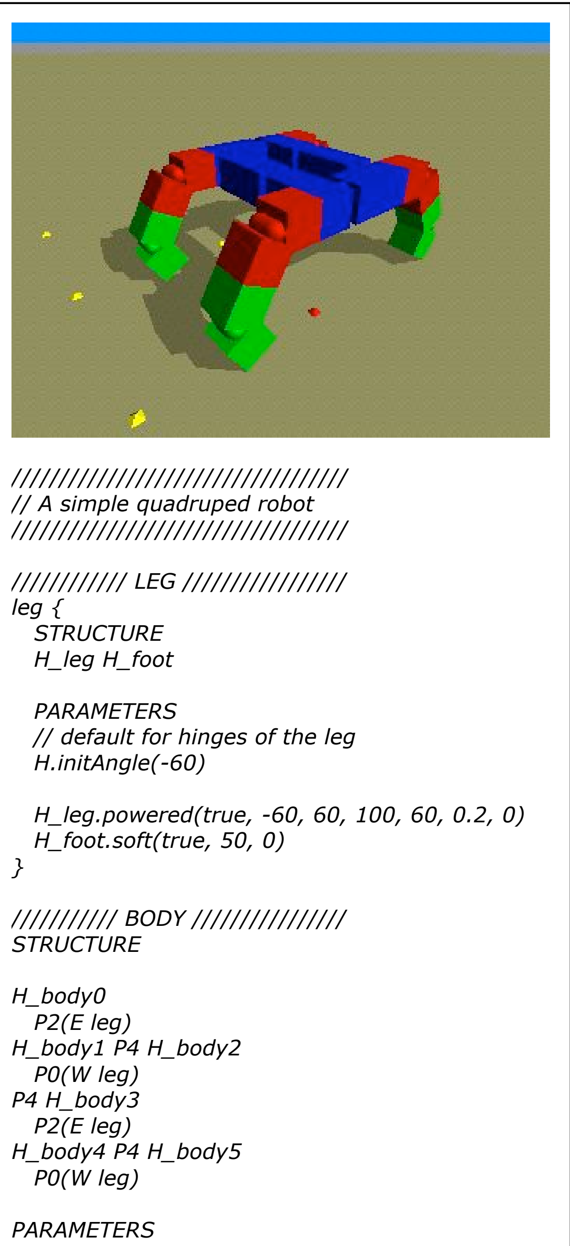
```
STRUCTURE
Hbody0 Hbody1
P4 Hbody2 P4 Hbody3
Hbody4 P4 Hbody5
PARAMETERS
```

Figure 12: This rectangle of rigid hinges will be the body of the quadruped robot. Note that Hbody0 and Hbody5 are not fixed together.



```
STRUCTURE

H_body0
  P2( E H_leg H_foot )
H_body1 P4 H_body2 P4 H_body3 H_body4
P4 H_body5

PARAMETERS

H_leg.initAngle(-60)
H_foot.initAngle(-60)

H_leg.powered(true, -60, 60, 100, 60, 0.2, 0)
H  foot.soft(true, 50, 0)
```

Figure 13: A leg has been attached at the right face of the first cube of Hbody0. The 'foot' is set to be elastic. Note that usually legs have two powered joints, for simplicity we didn't add a knee to this robot.



```
//////////////////////////////////
// A simple quadruped robot
//////////////////////////////////

/////////// LEG ////////////////
leg {
  STRUCTURE
  H_leg H_foot

  PARAMETERS
  // default for hinges of the leg
  H.initAngle(-60)

  H_leg.powered(true, -60, 60, 100, 60, 0.2, 0)
  H_foot.soft(true, 50, 0)
}

////////// BODY ///////////////
STRUCTURE

H_body0
  P2(E leg)
H_body1 P4 H_body2
  P0(W leg)
P4 H_body3
  P2(E leg)
H_body4 P4 H_body5
  P0(W leg)

PARAMETERS
```

Figure 14: The final script for the quadruped robot.

If you want to test your Adam scripting skills, finish the script of figure 13 by adding the three resting legs explicitly. Try to get the robot to walk. Set the phase of the first leg to zero and play around with the other three phases. Can you get it to jump by also modifying the other parameters of the powered hinges?

4 Genetic algorithm

This chapter describes the genetic algorithm (GA) that Adam uses to co-evolve morphology and control of robots. The primary goal of this project is the simulator of modular robots and not the evolution of a specific robot type. Consequently, the GA is rather primitive and probably not optimal, even though it has proved to evolve powerful solutions (see chapter X). Some of its basic parameters like the population size or the mutation rate, can be changed but a lot of other choices, for example how individuals are selected for reproduction and deletion, are hard coded and need further testing to know if they really perform well.

The implementation of this first version of a GA for Adam has two purposes. It should be the starting point for a more in depth analysis of co-evolution of modular, potentially even self-reconfigurable, robots. The second purpose is to test the stability of the simulation environment.

4.1 Phenotype space

The phenotype space of Adam robots is huge. Suppose you have a hinge *Ha* and you want to attach a second hinge *Hb* to it. If the two hinges are not yet attached to any other modules, there are a priori 400 ways to fix them together. *Hb* can be attached on any of the 10 free faces of *Ha* with any of its 10 faces, plus there are 4 possible ways of attaching two hinges together with two specific faces. A third hinge could be attached to any of the 18 free faces and this can be done, as before, in 10*4 different ways. Theoretically, only three hinges therefore offer already 1'120 different structures. This is only an estimate and not the exact number of structural configurations because some of them might be functionally identical, just built another way and it also has to be considered if the space that the attached hinges occupy is free. Nevertheless, this gives us an idea of the vast phenotype space we are dealing with. Note that we only considered how the hinges could be attached together; the parameters of the different modules are also part of the search space.

The precise definition of the Adam phenotype space relies on the following hypothesis:

1. *There's a finite number N of module types used (usually few).*
2. *There are an infinite number of modules available of each type.*
3. *There's infinite space available to build the robot (there are no limitations on size and form of the robot)*

The Adam phenotype space consists of all robots in the simulated world that could be built theoretically in the real world with corresponding hardware modules under hypothesis 1-3.

Note that phenotypes are simulated robots. The previous definition allows robots to have cycles but modules must not intersect because such a structure could not be built in the real world (it works perfectly well in simulation). For pratical reasons, , the second hypothesis can obviously not hold because the memory of the computer is limited and simulation gets very slow for robots with more than 50 hinges. Evolved robots usually had up to 30 modules.

On the other hand, hypothesis 3 must always hold: First, the robot is built in empty space, and then it is put into the simulation world (so there will never be such a thing as a leg entering into the ground).

4.2 Genetic Encoding

The type of encoding plays an important role in the evolution process, at the level of genetic operators, fitness landscape and, finally, speed and quality of evolution. The more complex a phenotype space is, the more difficult it is to find a good genotype-to-phenotype mapping, or genetic encoding. In nature, billions of genes are necessary to define organisms able to survive. Even though we have succeeded in decoding many genomes, we don't know much about the corresponding genotype-to-phenotype mappings.

There's an infinity of possible genetic encodings for a given phenotype space. An encoding and the corresponding genotype are complete if the genotype space is mapped *onto* the phenotype space. Recent studies on different genotype encodings for simulated 3D agents have found that completeness is not necessarily a good thing to have [1]. An incomplete genotype does not fully cover the phenotype space. It limits and structures the search space and therefore speeds up evolution, the drawback is obviously that some optima's might be lost because they're not covered anymore.

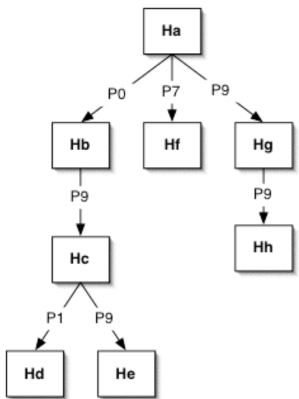


Figure 15: The Adam genotype is a tree. The head of the robot is the root; the nodes represent module instances with all their parameters.

A phenotype space has an inherent topology: phenotypes of similar fitness are 'close' to each other. The genotype encoding imposes another topology, by defining which phenotypes are close genetically (measured as the number of mutations separating the corresponding genotypes) and which ones have a higher probability of being visited. Under a good encoding, these two topologies are more highly correlated [5]. Unfortunately, this correlation is impossible to measure directly, because of the vast complexity of the phenotype space and its fitness landscape. Our judgments of encodings are thus subjective, based on various theoretical and experimental observations.

It is possible to apply the genetic operators directly to the phenotype itself without encoding it. This is usually done with simple, parameter optimizing GAs. For complex phenotype spaces, such direct encodings have proven to perform worse than higher-level encodings. The two most currently used genotypes for co-evolution of morphology and control of artificial creatures or robots are directed graphs and developmental encodings. The latter ones are based on L-systems and tend to result in more complex, better structured but not necessarily fitter solutions [1].

One could imagine using the script as genotype for Adam robots. However, the script has been designed to be easy to read and manipulated by humans and is not well suited for the GA. For example, the decomposition in a structure and a parameter part is convenient for users but very inefficient for a GA. To delete modules or to do crossover it would have to gather all the parameter-setting functions of the involved hinges in the second part of the script.

Adam uses trees, a subset of directed graphs, as genotype (figure 15). The main advantages of this representation are that it is compact, robust in the face of genetic operators and identical to the internal data type of Adam robots (see chapter 5.1), allowing the genotype-to-phenotype mapping to directly build the robot in the simulation world. With a developmental encoding this step would be much more complicated and time consuming. Unfortunately, it is impossible to express cycles and to reuse predefined body parts with trees alone. It would therefore be desirable to generalize the current genotype to directed graphs like the ones Karl Sims used (figure 3).

The genetic encoding structures the phenotype space. As mentioned above, the GA is suspected to perform better if this topology is highly correlated with the one induced by fitness values. The Adam genotype clearly fulfills this criterion because genetically neighboring individuals have similar phenotypes and therefore usually also similar fitness values. This is not true for developmental encodings where a single mutation can result in a completely different phenotype.

4.3 Genetic operators

The mutation operator acts on parameters as well as on the structure of the robot. The mutation rate determines the chance of a value being mutated. Its default value is 0.01 but it can also be set with the option *–mut* when Adam is launched. Random values are always selected with a uniform distribution.

The effect of mutation depends on the type of the value being mutated. Booleans are set to their opposite value while numerical parameters are reset to a random value, selected with a uniform distribution from an appropriate interval. The position and orientation that the module is fixed with can also be mutated and are set to another random position/orientation. Furthermore, each position where a limb can be attached has a chance of being mutated. If mutation occurs, a new, randomly initialized module is added if the position was free. In the case that there is already a limb attached at this position, it is deleted.

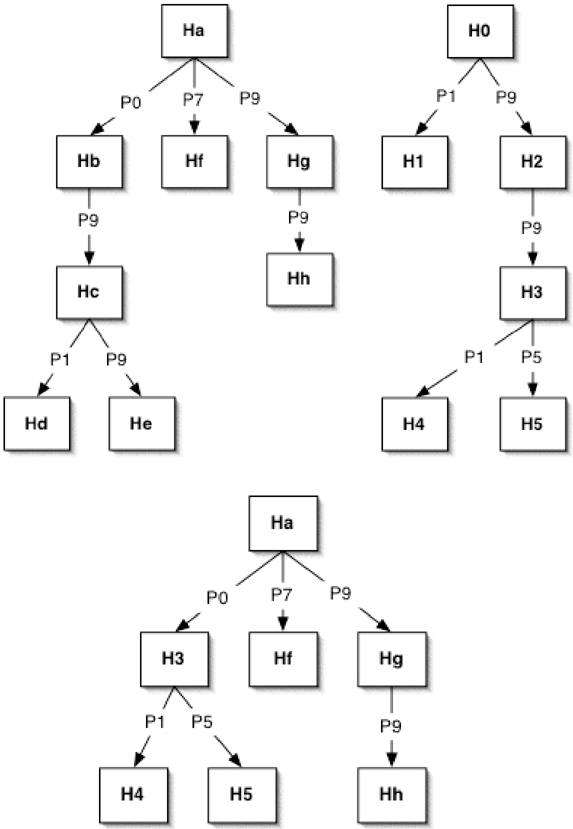


Figure 16: Mother (top left), father (top right) and child (bottom) after crossover. A randomly selected sub tree of the mother (Hb) has been swapped with one of the father (H3).

Thanks to the tree structure of the genotype, the implementation of crossover is straightforward. First a mother and a father are selected as described in chapter 4.5. The child is then formed by copying the mother and swapping one of its sub trees with a sub tree of the father (figure 16). The sub trees are selected by randomly choosing a node from the look-up table of the robot. Each module has an equal chance of being selected.

Obviously, crossover and mutation can generate invalid robots with intersecting modules. If this happens, the concerned robot is deleted and replaced with a new, randomly initialized individual. This is done in the hope of increasing diversity in the gene pool. In advanced populations, it might be better to replace such individuals with another child.

4.4 Initialization

At the beginning of the GA the population is initialized with randomly created robots. Individuals with self-collision are deleted and recreated until the whole population consists of valid robots. The size of the population can be specified when Adam is launched with the option –*pop*. It has a default value of 100 robots.

In order to increase the chance of creating valid structures, default positions have a higher probability of being retained. Experience has shown that better results can be achieved by not initializing with completely random values. For example, hinges have a higher probability of being rigid or powered than elastic, the frequency of powered hinges is always the same and the phase is a multiple of $\pi/6$. A hinge can only have up to two limbs and the depth of the tree representing the robot is limited at 3.

This initialization algorithm has proven to construct a big variety of mostly valid robots. It was surprising to see that very simple but efficient solutions were already found with the initialization, just by coincidence (figure X).

4.5 Selection and replacement

After evaluating a generation and assigning a fitness value to every individual, parents have to be selected to produce offspring. Fitter individuals have a better chance of being parents. This gives us selection pressure and drives the population forward. One should be careful that selection pressure is not too high; otherwise, the population diversity might suffer. There are many different strategies to select parents from the population.

Fitness proportionate selection (also known as the roulette wheel method) gives each individual a chance proportional to how good its fitness is with respect to the population fitness. Fitness proportionate selection does not work if all individuals of the population have roughly the same fitness. In that case, the chance of being a parent is almost the same for all individuals and selection pressure is too low. The other extreme occurs if an individual has a very high fitness compared to the rest of the population. Its probability of producing offspring is close to 1 and in a few generations, the whole population will consist of equal copies. For these reasons, fitness values should be scaled when using the roulette wheel method.

Tournament selection randomly selects *k* individuals. *k* is the size of the tournament. The best individual of the tournament is taken to make offspring.

Rank-based selection sorts individuals on their fitness values from best to worse. The place in this sorted list is called rank. Adam uses a rank-proportional roulette wheel method. The probability $p_s(i)$ for an individual i to be a parent is proportional to its rank r(i). N is the population size.

$$p_s(i) = (N + 1 - r(i)) / \sum r(i)$$

For every new child, another individual has to be deleted if the population size is to stay constant. To randomly choose an individual for replacement, the same methods as for selection of parents can be used. Now fitter individuals will have a smaller chance of being selected. Adam uses the same strategy for replacement as for selection. The probability $p_r(i)$ of an individual to be deleted is:

$$p_r(i) = r(i) / \sum r(i)$$

4.6 Algorithm

Adam uses a standard genetic algorithm. For every generation, a number of individuals are replaced with offspring. Afterwards the mutation operator is applied to all robots of the population except for the fittest one (steady-state evolution) and finally the fitness values of the modified robots are updated in simulation.

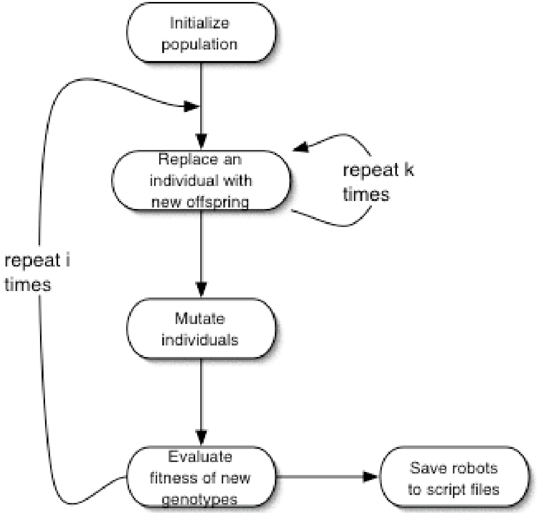


Figure 17: Diagram of the GA implemented in Adam. *k* is the number of replaced individuals per generation, *i* is the total number of generations.

5 Implementation

5.1 Internal representation of robots

Internally, a robot is represented as a tree (figure 16). Each module has pointers to the heads of its limbs. It is not necessary for the limbs to know to which module they are attached to so the tree is not doubly linked.

This tree structure has been proven extremely efficient, flexible and well adapted to represent Adam robots. The structural parts of the script are themselves representations of such trees (figure 15). The implementation of a recursive parser constructing a robot is therefore straightforward and efficient (chapter X). Furthermore, this representation can also be used as genotype. This simplifies the genotype-to-phenotype mapping a lot and therefore saves time in the EA (chapter 4.3). To speed up random access robots also have a lookup table with all their modules listed.

Note that in the script it is possible to point to predefined trees (body parts) from another one. The internal representation (data type) doesn't allow this. The body parts from the script are copied and added to the tree as many times as needed because currently one module represents exactly one object in the simulated world. This might change in the next version of Adam (see chapter X).

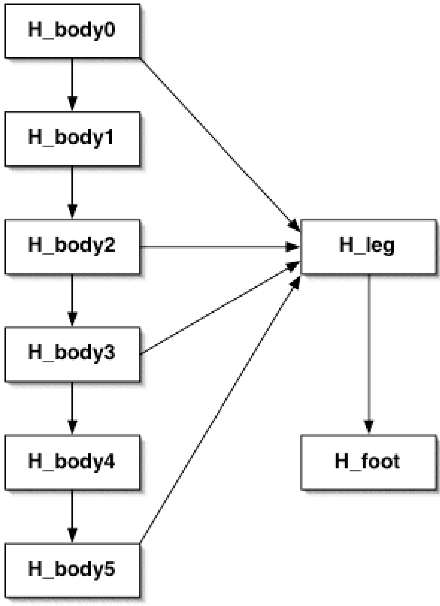


Figure 18: The structural parts of a script, in this case the quadruped robot of chapter X, describe the configuration as a set of trees. It is possible to point to previously defined trees (body parts).

5.2 Control of powered hinges

The desired angle θ of powered hinges is defined by a sinus oscillation as described in chapter 3.1:

$$\theta = A \sin(2\pi ft + \varphi)$$

But how can we achieve the desired behavior in simulation? The simplest solution is a PD controller. A couple of torques is applied to get a rotation of the hinge that depends on the difference between the desired angle θ_d, the actual angle θ_a and also on the angular rate ω (the derivative of the actual angle):

$$T = \alpha(\theta_d - \theta_a) - \beta\omega$$

However, ODE offers motors to control certain joints. These motors can be set at a specific speed (angular rate). This rate is applied immediately, in one simulation step. Even though it might seem weird to control the angular rate and not the angle itself, one should still use the motors and not apply torques directly because with motors ODE takes care that the simulation is stable. Therefore, control can be done by setting at each time step the angular rate of the motor to:

$$\omega = \alpha(\theta_d - \theta_a)$$

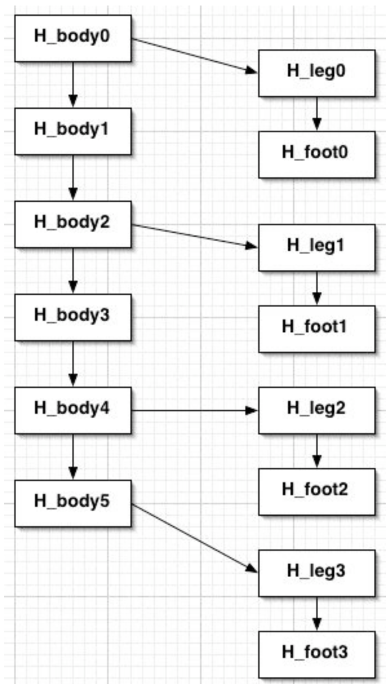


Figure 19: The tree of the quadruped robot from chapter X. This internal representation (data type) is also the genotype of Adam robots.

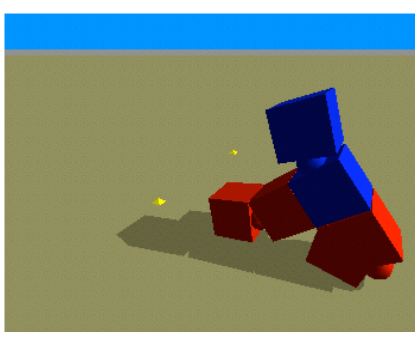
6 Results

Locomotion has been evolved with a very simple but effective fitness function. The fitness of a robot is defined to as its distance after a constant time of simulation from the origin. To calculate the fitness one only needs to know the x- and y-coordinates of the robot when simulation ends because the starting position is always in the origin. The position of the robot is defined to be the one of its head.

$$f = sqrt(x^2 + y^2)$$

This simple fitness function has proved to evolve better locomotion than more complex ones, considering the center of mass or the instant velocity of robots. An advantage is also, that robots are evolved to move less in a straight line and not just for example, turn in circles.

Experience showed that the time of simulation is crucial to achieve good results. If it is too long, the GA gets very slow but if it is too short, we don't really evaluate the ability of locomotion but only a fast jump at the beginning of simulation. Figures 20 to 23 show some examples of evolved robots.



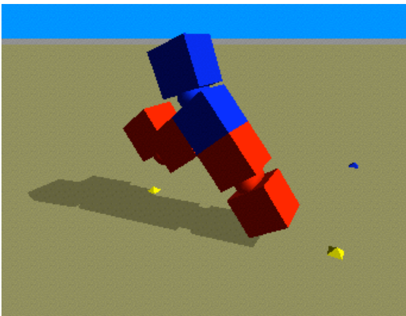
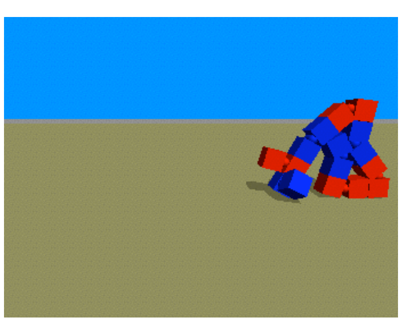
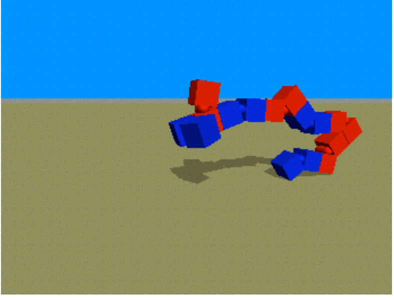


Figure 20: Simple but sometimes very efficient solutions were usually found in early generations. Later, they had no chance against more complex structures.





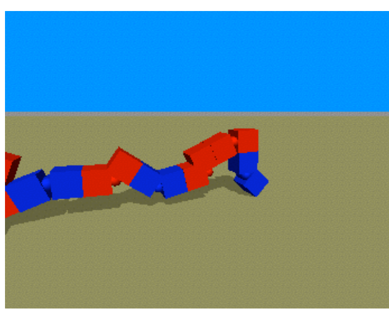


Figure 21: A jumping worm. This is (as well as the two following examples) a highly specialized individual that is difficult to improve on. Evolution took about three hours.

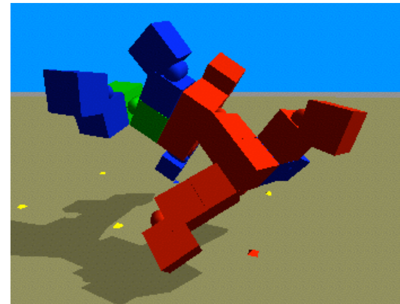
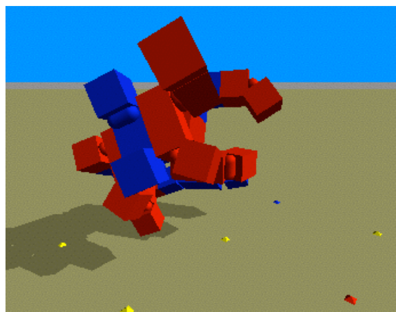


Figure 22: A two legged jumper. He has a long 'horn' on the front side of his head that effectively prevents him from falling over.



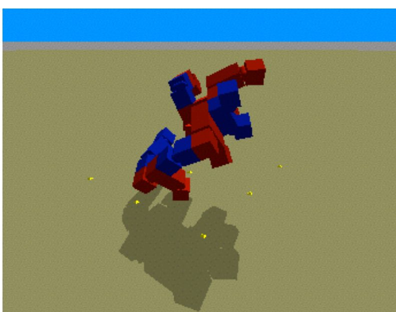


Figure 23: This dragon is one of the most feared creatures in the Adam world… It walks on two small legs keeping its balance with the arms.

7 Conclusions

The main goal of this project was to build a simulator for modular robots. By applying the genetic algorithm, we have tested this simulation environment repeatedly. I estimate the number of robots built and simulated by Adam well over a million – without a single crash. Furthermore simulation seems to be accurate and without bugs. Otherwise, evolved robots would have taken advantage of such irregularities. I also expected the simulation to perform well because ODE is a well-recognized simulation tool.

Clearly future work with Adam has to focus on the genetic algorithm and the control of the robots. It would be interesting to generalize the Adam tree genotype to directed graphs and to test developmental encodings. Furthermore, robots could be equipped with sensors and neural networks as controllers.

Another feature that should be implemented is self-reconfiguration. One could even somehow try to evolve self-reconfiguration. Obviously, cycles within the structure of the robot should be allowed as well.

References

[1] M. Komosinsky, A. Rotaru-Varga, Comparison of Different Genotype Encodings for Simulated 3D Agents. *Artificial Life Journal,* 7:395-418, 2001

[2] Hornby, Gregory. S. and Pollack, Jordan. B. (2001). Body-Brain Co-evolution Using L-systems as a Generative Encoding. *Genetic and Evolutionary Computation Conference.*

[3] M. Yim, D. Goldberg, A. Casal, Proc. of SPIE, Connectivity Planning for Closed-Chain Reconfiguration, *Sensor Fusion and Decentralized control in Robotic Systems III*, Volume 4196, Nov. 2000

[4] Karl Sims, Evolving 3D Morphology and Behavior by Competition. *Artificial Life IV Proceedings*, ed. by R. Brooks & P. Maes, MIT Press, 1994, pp28-39.

[5] W. Hordijk, Population flow on fitness landscapes.

[6] Jesper Rasmussen, A General Robot Simulator for Evolutionary Robotics. *Master's thesis*, 2001

[7] Biota website, http://www.biota.org/ksims/

[8] Modular Transformer website, http://staff.aist.go.jp/e.yoshida/test/top-e.htm

[9] Modular robotics at PARC website, http://www2.parc.com/spl/projects/modrobots/

[10] CONRO website, http://www.isi.edu/conro/

[11] Framsticks website, http://www.frams.alife.pl