Tuning physics based systems using Interactive Evolution

Kjartan A. Jónsson
University of Iceland, Department of Computer Science

October 2005

Abstract

This paper focuses on tuning physics based system using interactive evolution. To study this we choose to tune the physical properties of a simulated jeep in a game environment. The primary goal was to enhance user experience by tuning a limited set of attributes using two different approaches. One approached was primarily directed by the user while the other was more automated. Main conclusion drawn from this paper is that the benefit from using interactive evolution resulted in a larger attributes space being searched then when the user directly changed the attribute values himself.

1 Introduction

Interactive evolution can be used to direct the development of favorite experiences in various application areas where value of entertainment and progress is of more importance than real world simulation. Many programs today use physics packages to portray real life experience. These packages are used to combine primitive blocks to generate more complex physical representation of entities / objects. Each primitive block may then have one or more attributes that needs to be set for simulating the desired effect. Scale and position may be the most basic of attributes but depending on the package more complex values needs to be set. Common such attributes are mass, force and step.
As each primitive is combined with other primitives we end up with multiple of different attributes to be set for any one entity, and as each attribute is in some way dependent on other attributes we need to change one or more values when looking for the best user experience.
Most common way of tuning these parameters is to use the common user interface widgets provided by mainstream operative systems. Tweaking and altering numbers (either through some kind of widget like slide bar or manually changing numbers) until the numbers are reasonable in theory and then they are tested in the simulation world. Although feasible this method gets tedious when dealing with multiple physical building blocks that form a specific object like for instance a vehicle. For such systems where these blocks are interdependent a more effective way is preferred when searching for valid (and entertaining) values to simulate the behavior of the object.

Interactive evolution is an interesting approach to this problem because it alleviates the user from altering the values by hand (and mind) focusing more on weather the feeling of the behavior is correct rather than the numerical combinations. In systems like games where the entertainment value exceeds correct physical simulation it is an appealing approach to consider. Wolfgang Banzhaf stated, 1997, in his journal "Interactive evolution";

> *"In most of these cases, human intervention into the search and selection process would advance the search rather quickly and allow faster convergence onto the most promising regions of the fitness landscape"*

Benefits of interactive evolution rather then the numerical approach are that emphases are made on user experience. The tuner is constantly evaluating if the vehicle is behaving satisfactory for the specific purpose without having to evaluating the values being set.
Daryl H. Hepting (2003) explored IE and came to the conclusion that "*This new paradigm for acess to a large, complex information space has proven effective for exploration.*" ("Interactive evolution for systematic exploration of a parameter space") satisfying the criteria of most physical attribute spaces where these spaces are complex.
Interactive evolution has also been used within other fields in computer science with positive results.
Matthew Lewis and Richard Parent (2000) used it when evolving human figure geometry "*...producing any desired degree of realism and exibility of form, limited only by the prototype author's modeling skills…*" in their paper "An Implicit Surface Prototype for Evolving Human Figure Geometry"
Karl Sims (1991) concluded that "*Artificial evolution has been demonstrated to be a potentially powerful tool for the creation of procedurally generated structures, textures, and motions*"
Using this approach for computer animation Ik Soo Lim and Daniel Thalmann (2000) stated that "*It is potentially applicable to a wide variety of search problems, provided the candidate solutions can be produced quickly by a computer and evaluated quickly and easily by a human.*"

Is interactive evolution a feasible approach for tuning physics systems such as games? Could it enhance user experience? Evident that it is a step in the right direction would be if the attributes converge to the same values independent of start values.
Is this the case? Then how much iteration do we need to find the preferred values and could this be done with limited user participation?

This paper begins by examining the problem and what needs to be considered when working with these attribute values. Then the paper continues to evaluate different search methods scanning the attribute space. The two different approaches evaluated are "refinement based on user request" as suggested in section (3.1) and "refinement based on user performance" in section (3.2).

2 Tuning attribute values

The values we attempt to search are a part of the physics attribute describing / driving the behavior of a set of objects. What we do search for are attributes such as mass, force, bounce, softness, etc. Where each of these attributes may contain one or more variables that need to be set. Common attributes that we do not alter during our experiments are position, orientation and scale of primitive blocks within the entity. This would affect the topology.

How do we best evolve these values? Since the attributes reside in the attribute space defined by the packages then searches these values in the context of our simulation world.

Changing the attributes using a user interface (UI) is the simplest and most straight forward way. But using a UI we are forced to change between the UI and the simulation world gets quite cumbersome. As Ik Soo Lim and Daniel Thalmann (2000) stated in their paper "Solve Customers' Problems: Interactive Evolution for Tinkering with Computer Animation" that "*Manual parameter tweaking is, however, very tedious due to the multi-linearity, nonlinearity and discontinuity of the mappings between the input parameters and output values*"

Allowing the user to alter the values himself may also have a psychological impact as well where the user expects a change in proportion to the change he made. This may not always be the case. With this in mind I did not want the users to be able to tweak the values themselves.

Could we attempt to change the values on a per attribute basis? Since the attributes depend on each other changing one attribute per iteration this may result in the users need to tweak other dependent values to find the best solution. Altering on a attribute basis also is more time consuming. As a result we choose to alter a set of dependent attributes per iteration.

Finding the best experience from a set of experiences we must try to address two problems. First problem is that the concept experience is abstract and thus hard to measure. When dealing with more than one attribute we have no way of having an overview of previous experiences making it hard for the user to compare and select the preferred one.

Attempting to address this problem we measure the user's performance in our simulation and use them as reference points. Measurable values are dependent on the application and its usage; measurable performances are elapsed time, stability, average speed. Comparing these reference points with each other we choose the set of attribute-values that give the best preferred experience.

3 Solution

We can search the attribute space for appropriate values using interactive evolution (IE) where human interaction drives the direction of the evolution.
All evolutionary algorithms use selection. Interactive evolution (or Aesthetic Selection) is a form of evolutionary strategy but relies on user feedback for its evolutional method of selection.
Evolution strategy uses primarily real-vector coding, mutation, recombination, and selection as its primary operators. Mutation is performed by adding a Gaussian distributed random value.
Since the fitness function is based on the user preference then IE could be an appropriate way to simplify this kind of attribute-value searching.

We start with best known value and tune them per iteration. Using appropriate sigma (σ) we generate a Gaussian distributed random value around our last best known attribute.

$$\pi_n = Gauss(\sigma_n)$$
$$y_n = x_n + \pi_n$$

Where x_n are our attribute and σ_n our sigma that generates a distributed random value using Gauss around x_n with range of σ_n, generating new attribute value y_n.
The sigma (σ_n) value is predefined by the user based on the effect and influence on the other attributes. It is important to work with a correct sigma value so that the change is even among all attributes per iteration. If the sigma value is to large then the attribute may obscure the effect of all the other attributes resulting in only fitting that value to the fitness function. Also the opposite effect must be avoided so that attributes changes to little resulting minor changes to the overall effect.

Since the experience concept is abstract and when we only have the last experience as reference we choose to implement two different approaches. First one where the user requests alterations (3.1) is based on the user being able to evaluate and compare between the experiences.
The second implementations we use a measurable point of reference that helps search the attributes space. This should be measurable and describe the user's performance. Using this "knowledge" we attempt to help drive the evolution (3.2).

3.1 Evolution based on requests

We drive the evolution in such a way that the parent generates an offspring per attribute using $(1+1)ES$ method. This offspring is then used as the current attribute value to be evaluated by the user.

$$y = x + Gauss(\sigma)$$

If the user evaluates the offspring y to perform better than the parent x then the new generation value is assigned as the parent and a new offspring is generated.

```
while user requests tuning attributes

      Generated values for simulation space
      For each attribute in a collection of Attributes
            attribute_eval = gauss (sigma, attribute_best)

      Test the newly generated value in simulation space
      result = Drive the simulation

      Request feedback from user
      if result better than last simulation
            attribute_best = attribute_eval
```

Figure 1: Pseudo code implementation of (1+1)ES

Generations converge to fit user request where the user is the fitness function evaluating each generation.

The user evaluates each new experience with the previous and decides if it has a higher experience value than the previous based on his recollection. Figure 1 contains a pseudo code implementation that describes the process, step by step:

3.2 Evolution based on performance

By using some kind of system to measure performance we can compare new attribute-values using batch learning methods or $(1+\lambda)ES$ where lambda represents multiple offspring's. We simply generate new offspring values per attribute as above and try to accomplish some specified task. As this task is concluded we store the attribute values along with the result. Then after some n iterations we evaluate many different possible offspring's from the same parent and select the best performer by the fitness function (the user).

$$\begin{aligned} y_1 &= x + Gauss(\sigma_1) \\ y_2 &= x + Gauss(\sigma_2) \\ &\vdots \\ y_n &= x + Gauss(\sigma_n) \end{aligned}$$

Thus a new generation y_n is generated based on the previous parent x, and the best performer replaces the previous parent.
We continue to evolve the attributes based on the best performer of a population of offspring's through many generations and stop either when the fitness function is satisfied or little changes occurs to the attribute set.

```
For each evolutionary_step in a evolution

    For each iteration i in a evolutionary_step

        Generate values for simulation space and save them*
        For each attribute_eval[i] in a collection of Attributes
            attribute_eval[i] = gauss (sigma, attribute_best)

        Test the newly generated value in simulation space*
        result[i] = Drive the simulation

    Get the best results iteration index
    iteration = argmax (result)

    Set the best attribute values as final values, for next iteration
    For each attribute_best in a collection of Attributes
        Attribute_best = attribute_eval[iteration]
```

Figure 2: Pseudo code implementation of (1+λ)ES

Comparing the pseudo code in figure 1 and 2 the only reused code sections are marked with a star '*'.
(See appendix C for implementation using python)

4 Implementation

By using the OGRE graphics framework (Object-Oriented Graphics Rendering Engine) and ODE (Open Dynamics Engine) through OGREODE (Object-Oriented Graphics Rendering Engine - Open Dynamics Engine) an implementation using C++ and Python where created to generate a vehicle simulation engine.

We choose to find the feeling of our simulation by evolving the attribute settings of a vehicle. The vehicle consists of a body and four wheels attached to two separate axes. Primitive blocks consisted of bodies, joints, hinges, sliders and angular motors.

Refereeing to figure 3 we see a vehicle assembled from primitive building blocks. Where each building block may require one or more attributes to be set.

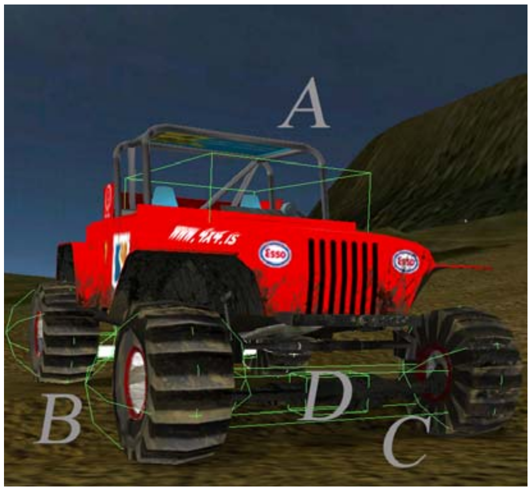


Figure 3: The trained vehicle. Green line surrounding all items representing a primitive building blocks

A. Chasid consists of a body
B. Wheels consists of angular motor and hinge-2 joint that is attached to C
C. Axis consists of a body which is attached to D using a hinge joint.
D. Hinge is attached to A using a slider to represent the suspension system.

Using OGREODE the following attributes had to be set to generate the vehicle in our simulation world.

Name	Group set	Batch group	Sigma	[Min, Max]
Fudge Factor	Wheels	Control	0.1	[0,1]
Power Factor	Wheels	Control	0.1	[0,1]
Brake Factor	Wheels	Control	0.1	[0,1]
Hinge Spring	Hinge	Stability	50	[0,1000]
Hinge Damping	Hinge	Stability	0.1	[0,1]
Hinge Bounce	Hinge	Stability	0.01	[0,1]
Slider Spring	Slider	Stability	7	[1,150]
Slider Damping	Slider	Stability	0.1	[0,1]
Slider Bounce	Slider	Stability	0.01	[0,1]
Mass	Chasid	Stability	1.5	[1,100]
Suspension Spring	Suspension	Stability	50	[100, 900]
Suspension Damping	Suspension	Stability	0.05	[0.4, 1]
Suspension StepSize	Suspension	Stability	0.003	[0.01, 0.05]

Table 1: Attributes that where tuned and their groups

To evaluate if using this method was a viable way to search the physics based attribute space we wanted to examine a large attribute set versus a small concentrated set. The "Group set" is the attribute values for a specific building block. The "Batch group" forms a collection of attributes that changes a specific behavior.

The terrain used to drive the vehicle is uneven and rugged to simulate rough courses. Using this terrain the application stabilizes the vehicle to an extent where the user can control the vehicle when driving in high speeds.
The system stored the setup of the vehicle in an Extensible Markup Language (XML) file including the attributes-values the trainer is searching. In the beginning of the session the default values are set when the car is loaded. During training the best values where saved into this same file.
In this experiment all different aspects of the solution was examined by learning both singe attributes and multiple sets of attributes at a time. For each of the two approaches (evolution based on request and progress) I choose to focus on different number of attributes per set.
The "Group set" tunes the vehicle by focusing only on specific characteristics of the vehicle. For instance the "Slider" group only trains the slider setting spring, damping and bounce values. "Batch group" on the other hand groups the "Slider" values together with other attributes forming the group "Stability" which controls the how much it adjusts to the ground when driving the vehicle.

3.1.1 Evolution based on request

Training these groups using evolution based on requests started of by using manually selected random values. Then the trainer selected which set of group he wished to train (wheels, brakes, hinge, slider, chasid, suspension, control, stability) using a combo box containing these groups. The trainer then requested alterations on demand until satisfied using a button within the application. When satisfied the trainer saved the result (to file). Successive requests where then based on these stored values. The training continued throughout the session until the trainer found the optimal result. Successive sessions continued the same way; beginning by altering only a few randomly selected start values.
The trainer began at some random place in the terrain and could drive around at his own leisure and train the specified group. There was neither a time limit nor specific goals to accomplish.

3.1.2 Evolution based on performance

Training using evolution based on performance started of by selecting one vehicle setup file that was going to be the only start values for every session. Evaluating each session would end with the same attribute-values within each set.
Each session began by the user specifying what group the trainer wished to focus on then the vehicle appeared at the start gates. A count down began from three and when it reached zero the user should start. When the vehicles back wheels moved cross the start gate the timer began. The goal was to reach the end gate as fast as possible across a lightly bulgy landscape where the last ten meters towards the end gate had a hill that the vehicle had to maneuver. The course began with a 180 degree turn and the hill was three times the size of the vehicle that needed to be climbed.
When reaching the end gate the user was notified of the time that it took to reach the gate, then after some seconds later the vehicle appeared once again at the start gate for a new lap.
Each iteration contained ten laps where the only interaction by the user to the system was driving the vehicle. A training session consisted of 10 iterations per attribute group.

4.1 Results

Evaluating weather interactive evolution is valid way to search the attribute space of physical entities we decided to form group sets that we have some idea on how they would perform. The “control” group should have attributes with highest valid values. Although power- and fudge factor, which describes the amount of power output for a wheel and how well the wheels grip the surface, have close dependency of each other, the brake has none with the relation of the two. “Suspension”, “hinge” and “slider” groups contains attributes that heavily are dependent on each other. The “stability” group all support each other stabilizing the vehicle on uneven ground.

4.1.1 Evolution based on requests

Tuning a vehicle using interactive evolution based on request from the user was simple and straight forward. Often the values were acceptable but it did occur that some value combinations where generated that was not valid, due to the independency of the attributes being set. Some attributes got bad values that in conjunction with other that influenced negatively in the overall setup.

It was also hard to know when the best values for the attributes where found. This occurred even when limiting the search space to a specific behavior. When we found a good value but decided to continue searching; we some times ended up with lesser values multiple times in a row. Then when we found a value that was slightly better than the lesser ones then we promptly accepted. This did not guarantee that we got better experience than the one we first found, only that it was best of the lesser ones. Eiben and Smith (2003) wrote that “*EAs are stockastic and mostly there are no guarenties to reach an optimum.*”

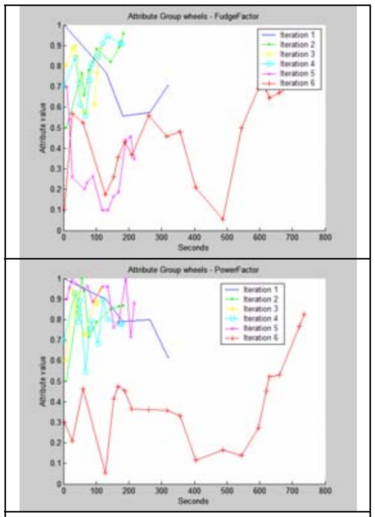


Figure 4

An example of this was when finding the best value for wheel performance. We limited the search space to include FudgeFactor (which describes how well the wheel grips the surface) and PowerFactor (the amount of power output for a wheel) which was changed simultaneously for all wheels. Examining the line chart “Attribute group wheels – FudgeFactor” (chart nr A.15 in appendix A) and “Attribute Group wheels – PowerFactor” (A.16 in appendix) in figure 4 , we see that we start with good values and end up with a lesser values for those attribute spaces. Unless we started of with attributes values that perform badly (as in iteration 6) then we see that they converged to better performing values.

Knowing when to stop turned out to be hard as well. When satisfied with the vehicles behavior it was tempting to continue searching the attribute-space with the risk of ending up with lesser performance. First iteration when searching “Batched Attribute Group control” (A.01-A.03 in appendix A) shows continuing to evaluate the space for ten minutes ending up with lesser values (for the FudgeFactor) or similar values (PowerFactor and BrakeFactor).

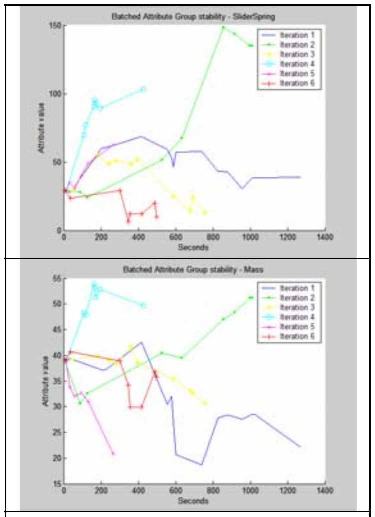


Figure 5

At occasions it was time consuming searching the attribute space for the correct values. Where searching a limited space: as in "slider" group set (in table 1) it took approximately five minutes to find the optimal values. Searching a larger space like the value space of "stability" often took twice as long.
The benefit of searching the larger space was more apparent though. We ended up searching a larger range of values and the multiple combinations of them. The values converged to extremes that would not be considered at first by the trainer. Example of such a change was when searching "Batched Attribute Group stability – SliderSpring" (A.10) figure 5 which went to much larger values during iteration 2 and 4 due to increase of vehicle mass in "Batched Attribute Group stability – Mass" (A.07).
It was important to combine attributes together that supported or where dependent on each other. When searching "Batched Attribute Group control" (A.01-A.03) we combined PowerFactor, FudgeFactor and BrakeFactor. The later was independent on the two first which ended up with both PowerFactor and FudgeFactor performing less than recommended after last iteration due to the trainers focus on the BrakeFactor attribute.

4.1.2 Evolution based on performance

This method was more promising than "evolution based on request". The vehicle ended with better user experience for the task that we wanted it to perform. For every iteration it formed a better to the fitness function for the defined task.
We trained the vehicle during 10 iterations by driving each course ten times for iteration. Each lap generated new attribute values from the last best known values and after ten laps the best performer values was set as the parent values to generate new values for next iteration.
In this case we wanted the user to be able to control the vehicle in high speeds over the defined terrain. The attribute values converged slower than using the two previous methods and kept within reasonable limits.
When searching the space for best performance attribute values in high speeds we notice that during the tuning of batched group stability; SuspensionSpring (B.05), SliderSpring (B.06), SuspensionDamping (B.04) and SliderDamping (B.07) got larger values while the mass decreased shown in figure 6. All forming better set of values for high speed trails, concluding that a better fit was formed. The shortest recorded time was approximately 20 seconds to complete the course but ranged from 30 to 40 seconds on average.

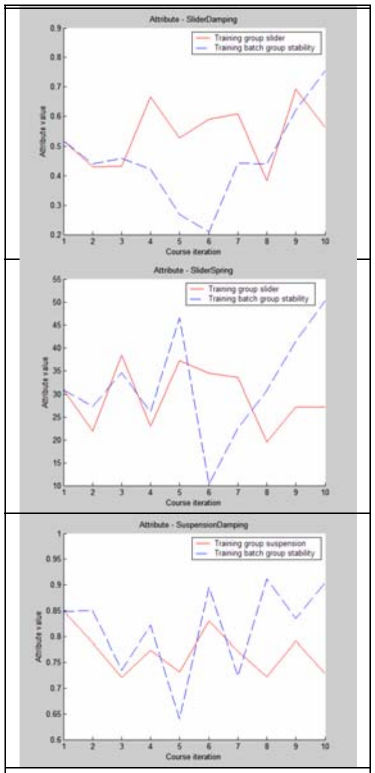


Figure 6

5 Conclusion

Whether the game developing community is using interactive evolution for tuning games is not evident since no publications are on the subject. However using interactive evolution to tune a physical entity is a valid choice. With this method the entire attribute space is evaluated and having many attributes being grouped examines different possible combinations.
Tuning the values manually may not take all value combinations into consideration since they may not be reasonably acceptable, resulting in only some values being tested by the tuner. Tuning with this method the trainer also has no pre determined expectancy of the values being evaluated since it is not displayed to the user during the training. This is important when determining the best experience so all possible scenarios is considered.

From the two approaches considered, the second approach; using a measurable point of reference performed better. The vehicle converged to handle the specific task that it was set out to adapt. Using the second approached though was more time consuming and required much more work rather than "evolution based on request".

Using interactive evolution we found that working with smaller attribute groups provided the best result. A large sized group requires more iteration for finding the optimal solution. For each added attribute the number of possible combinations of large versus smaller values grows exponential.
The grouped attributes should also share the same effect. Resulting in altering a group of attributes alters a common experience.
The number of iterations should preferably be kept to a minimum, as Wolfgang Banzhaf (1997) puts it "*... in sequence, only a limited number of generations can be practically inspected by a user before the user becomes tired.*"
Driving the vehicle for much iteration gets tiresome and resulted in mistakes being made during the training process. On the other hand the trainer got used to the terrain and knew better how to maneuver the vehicle to accomplish the best times.
A course that takes half a minute to complete ten times for iteration gets time consuming and wearisome for the trainer to evaluate. Having the trainer (driver) performing his/hers best is important since finding the best values for a vehicle.
When doing these experiments we did not expect to get blinded by less performed vehicles performance resulting in the trainer selecting the best of lesser ones as the optimum attribute values for each session. To avoid this, a better method would be to store all setups that the trainer is satisfied with and as a last step evaluate between these setups for the optimal setup. Also allowing the trainer to switch between the last optimum setup the trainer was satisfied with and current would alleviate this problem to some degree.
Searching the attribute space was effectively accomplished even though we are not sure that the optimal values where found.

The attributes did not converge to the same values independent on the start values as we first expected. Is it that many different combinations of values may portray the same user experience or is it due to the difficulty of evaluating between the different experiences? The reason is not evident and requires further research.

References

Wolfgang Banzhaf, 1997, "*Introduction to Evolutionary Computing*", *Handbook of Evolutionary Computation*, vol 1, IOP Publishing Ltd and Oxford University Press

Daryl H. Hepting. 2003, *Interactive evolution for systematic exploration of a parameter space*. University of Regina

Matthew Lewis and Richard Parent. 2000, *An Implicit Surface Prototype for Evolving Human Figure Geometry*. Technical Report OSU-ACCAD-11/00-TR2, Page 1

Karl Sims, 1991, *Artificial Evolution for Computer Graphics*. ACM Computer Graphics, pp. 319-328.

Wolfgang Banzhaf, 1997, *C2.9 Interactive evolution*, IOP Publishing Ltd and Oxford University Press, page 2

Ik Soo Lim and Daniel Thalmann, 2000, *Solve Customers' Problems: Interactive Evolution for Tinkering with Computer Animation*, Page 2

A.E. Eiben and J.E. Smith, 2003, *Introduction to Evolutionary Computing*. Springer. Chapter 2, page 23.

Bibliography

Wolfgang Banzhaf, 1997, "*Introduction to Evolutionary Computing*", *Handbook of Evolutionary Computation*, vol 1, IOP Publishing Ltd and Oxford University Press

Wolfgang Banzhaf, 1997, *C2.9 Interactive evolution*, IOP Publishing Ltd and Oxford University Press

David M. Bourg, 2001, *Physics for Game Developers*. O'Reilly.

A.E. Eiben and J.E. Smith, 2003, *Introduction to Evolutionary Computing*. Springer

Daryl H. Hepting. 2003, *Interactive evolution for systematic exploration of a parameter space*. University of Regina.

Tom M. Mitchell, 1997, *Machine Learning*. McGraw-Hill Science/Engineering/Math

Karl Sims, 1991, *Artificial Evolution for Computer Graphics*. ACM Computer Graphics.

Russell Smith, may 2004, "Open Dynamics Engine manual v0.5.", online handbook, viewed 2005, <www.ode.org>

Bjarne Stroustrump, 1997, *The C++ Programming Language (3rd Edition)*, Addison-Wesley Professional

Object-Oriented Graphics Rendering Engine, 2005, *OGRE 3D : Open source graphics engine,* viewed october 2005, <www.ogre3d.org>

OGREODE, 08:13-17 Jun 2005, *OgreOde - Ogre Wiki*, a C++ wrapper for ODE above the OGRE framework, viewed 5 october 2005, <www.ogre3d.org/wiki/index.php/OgreODE>

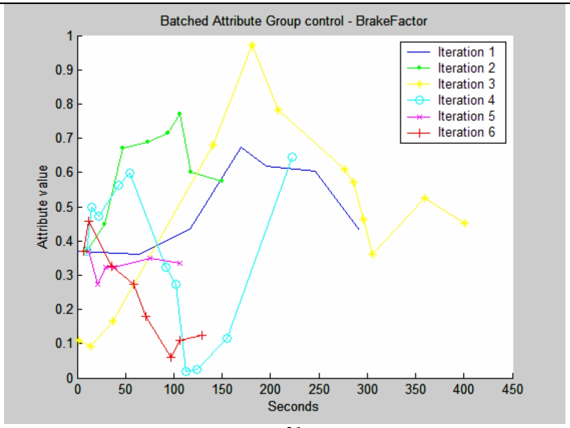
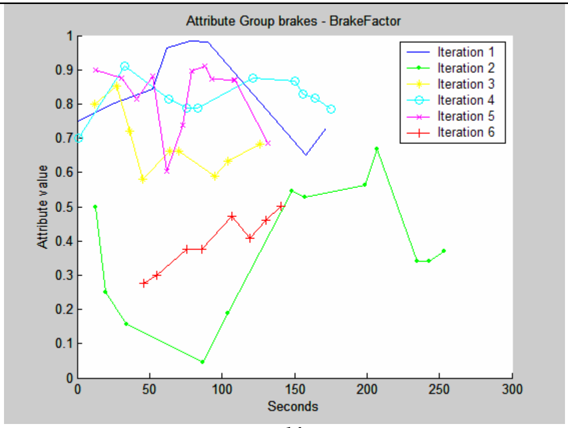
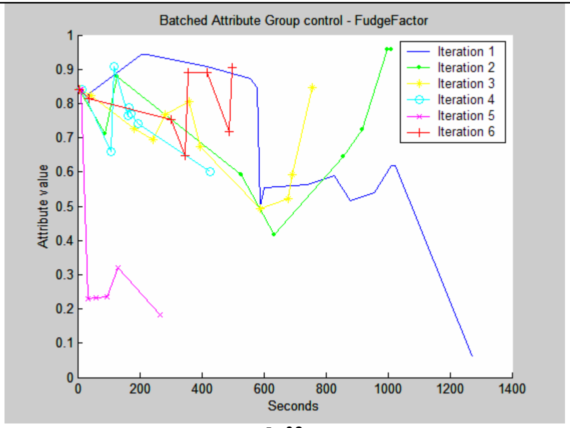
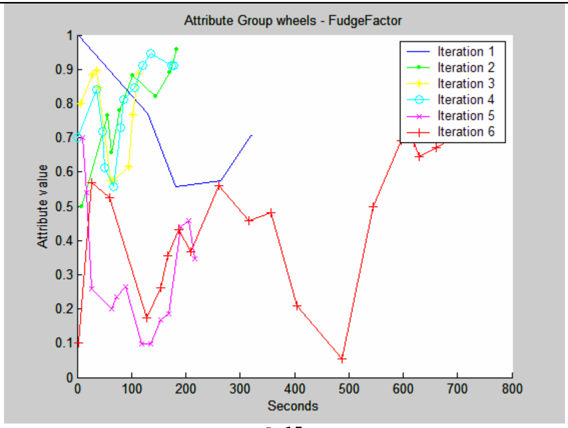
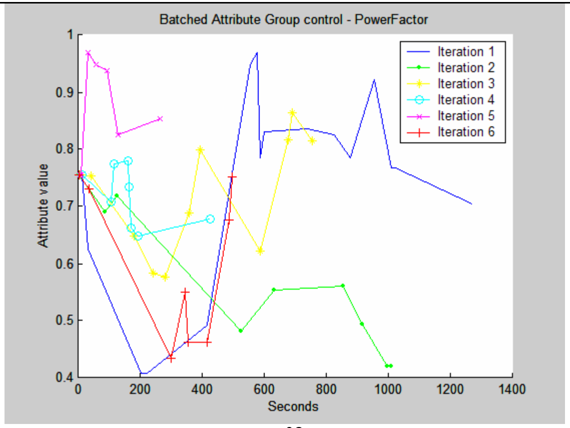
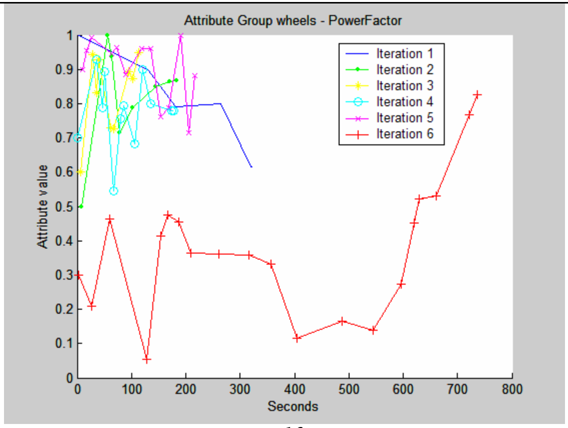
Ik Soo Lim and Daniel Thalmann, 2000, *Solve Customers' Problems: Interactive Evolution for Tinkering with Computer Animation*,

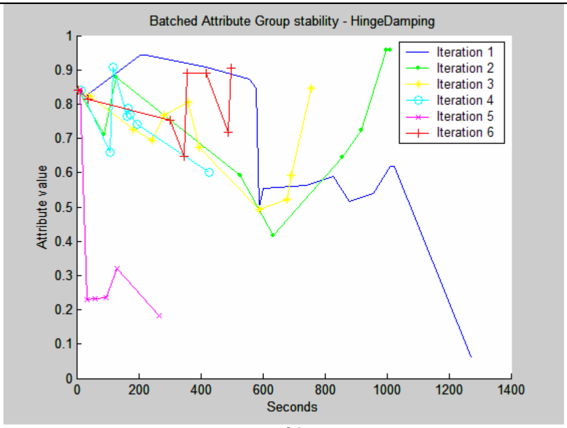
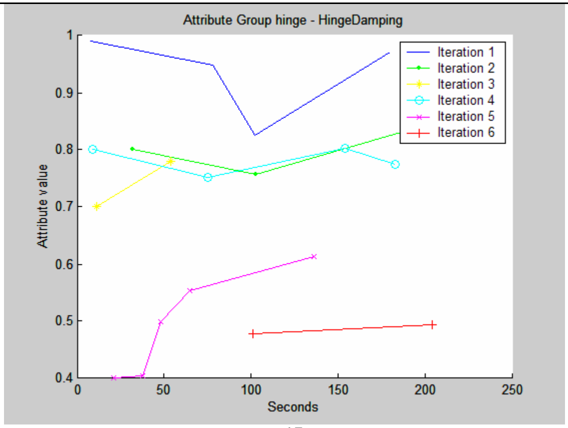
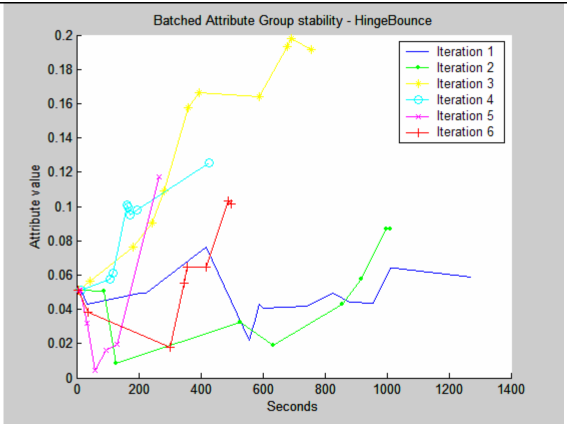
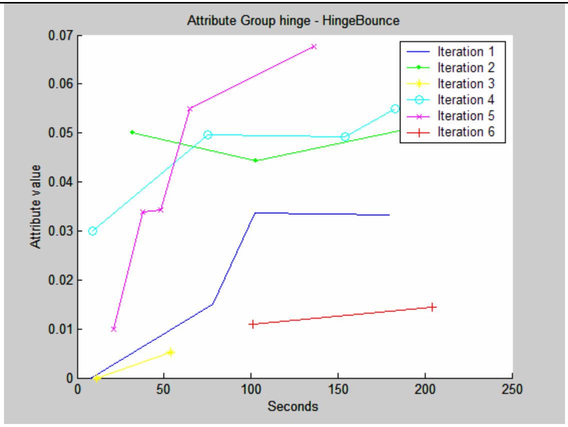
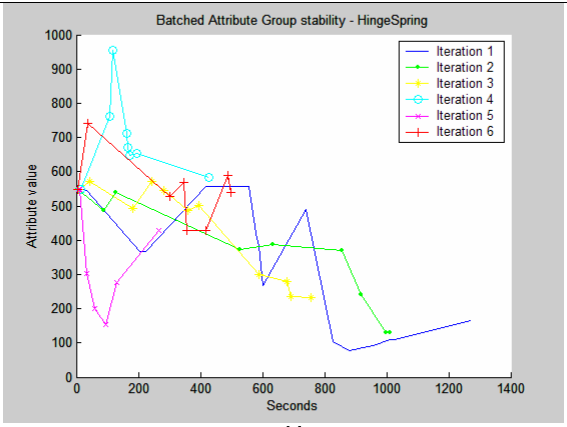
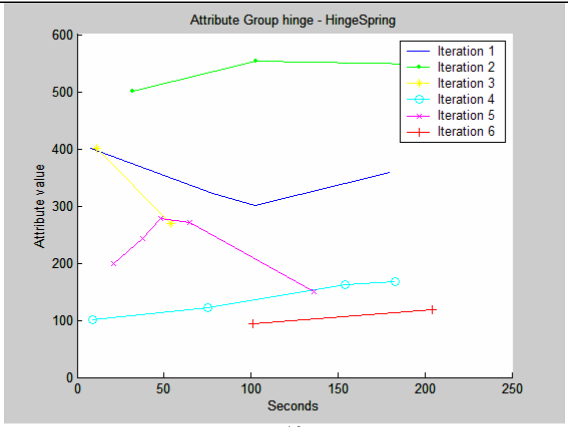
Matthew Lewis and Richard Parent. 2000, *An Implicit Surface Prototype for Evolving Human Figure Geometry*. Technical Report OSU-ACCAD-11/00-TR2

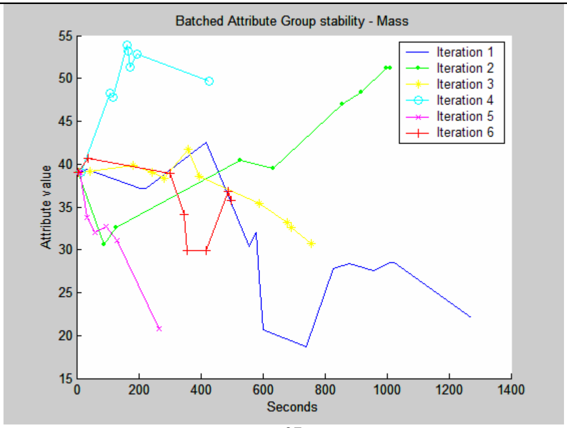
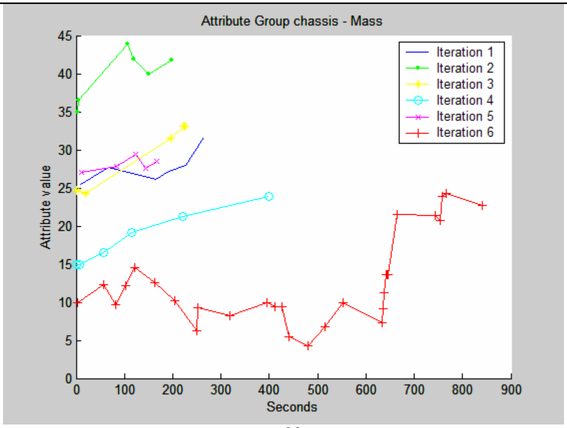
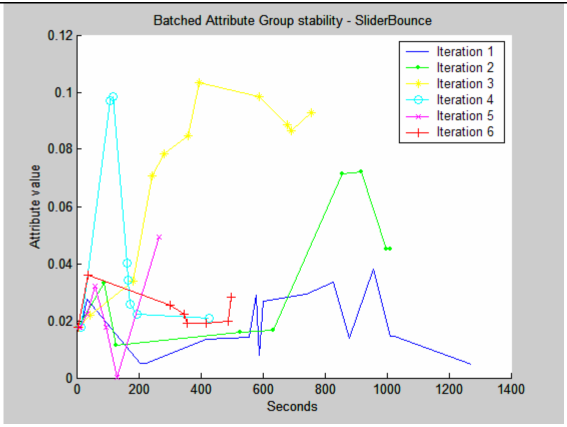
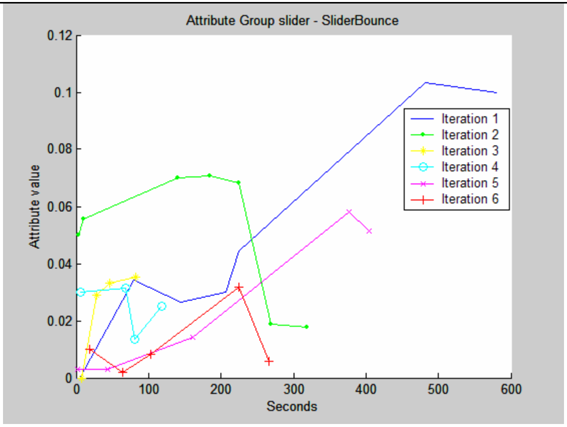
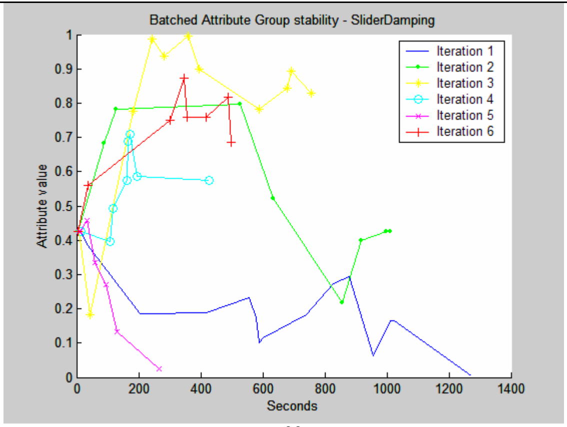
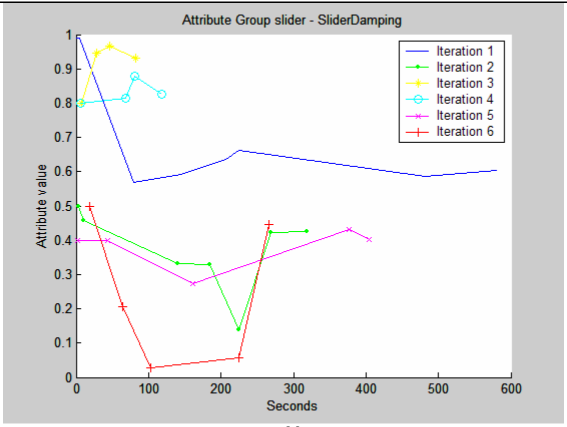
Python, 2005, *Python Programming Language*, an interpreted, interactive, object-oriented programming language, viewed 2005, <www.python.org>

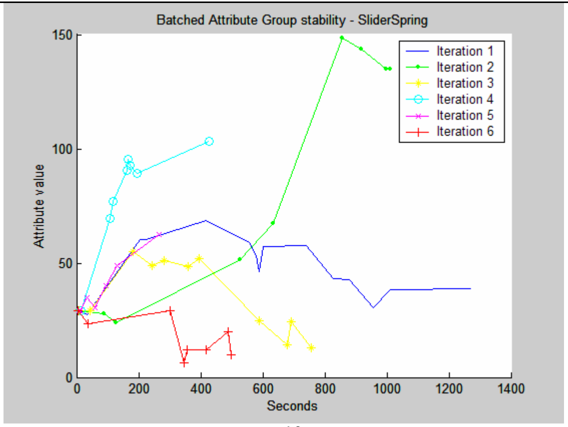
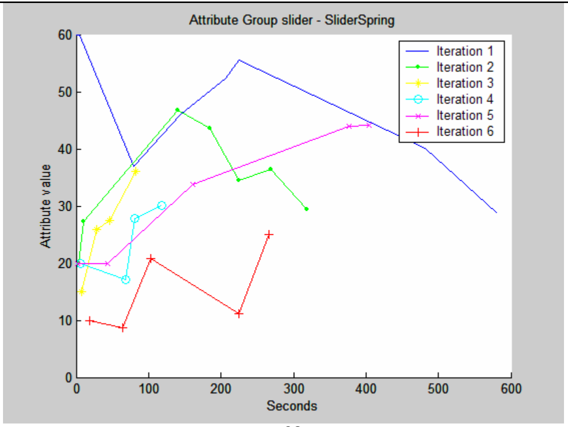
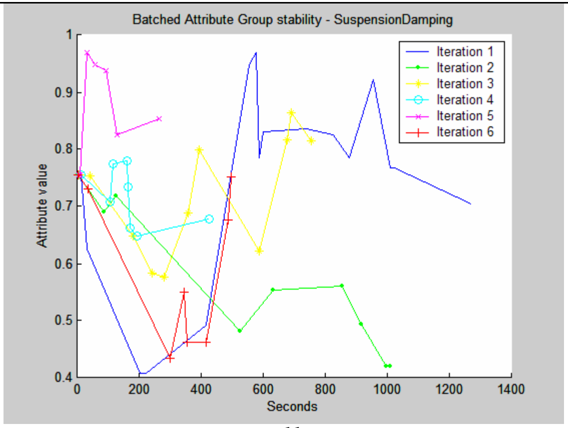
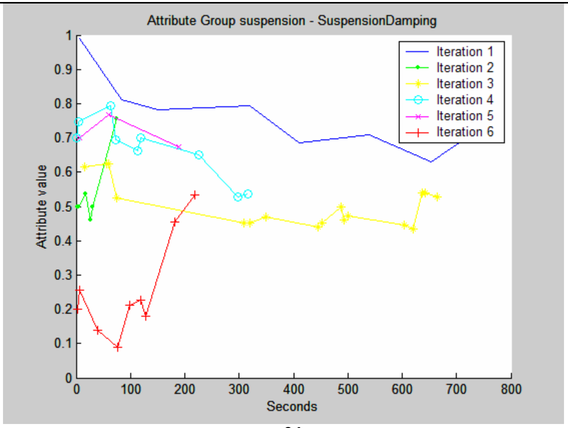
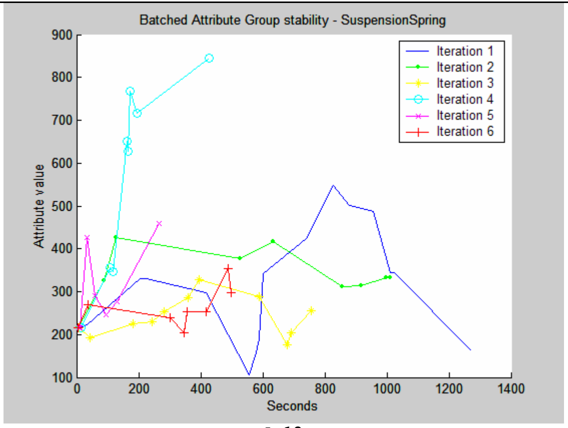
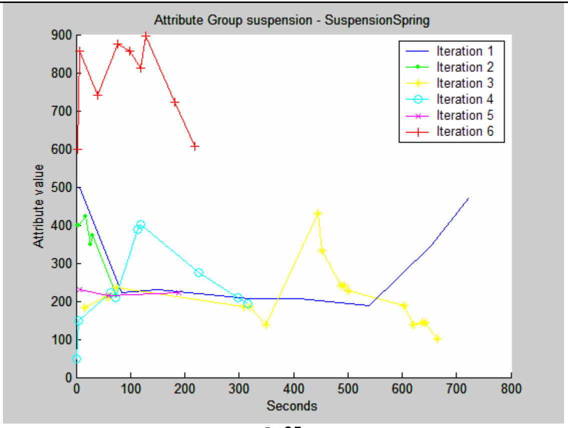
W3C Recomendation, 04 February 2004, *Extensible Markup Language (XML) 1.0 (Third edition)*, viewed october 2005,<www.w3.org/TR/REC-xml/>

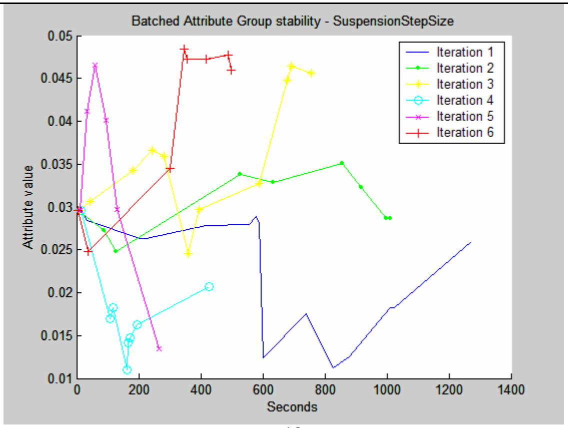
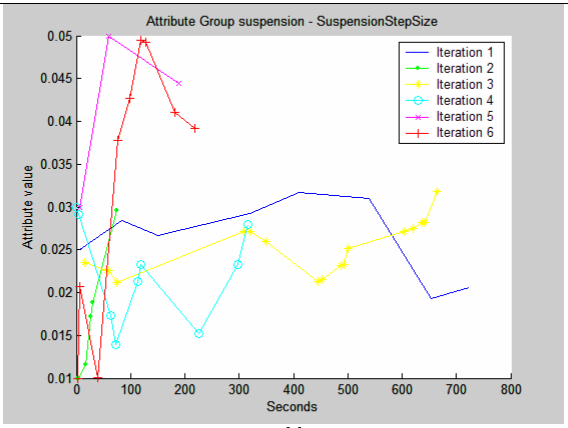
Appendix A: “Evolution based on request” charts

Brake-Factor The amount of brake force. Since the brakes where in the same group as fudge- and power factor (A.01) we see that on iteration 4 it reached very low values. The trainer seems to focus on the other two attributes at that time.	 A.01	 A.14
Fudge-Factor The amount of traction force that is available. Both columns of graphs displays how each attribute progresses. The difference is in what group it is trained. Throughout all these graphs we see little correlation between the columns.	 A.02	 A.15
Power-Factor The amount of power available in per	 A.03	 A.16

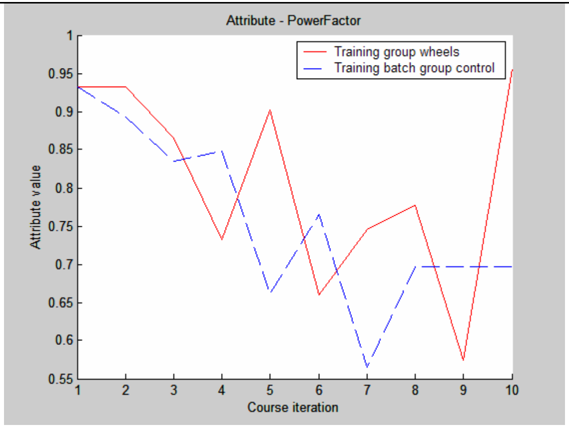
Hinge-Dampig The damping effect on Hinge-Spring. Training Hinge-Damping for “Attribute Group hinge” (A.17-A.19) gave little feedback to the trainer. This is why these diagrams are scarce on information	 **A.04**	 **A.17**
Hinge-Bounce The bouncy effect on hinge	 **A.05**	 **A.18**
Hinge-Spring How stiff the spring is to the weight it is carrying. Comparing A.06 and A.07 for iteration 4,6 and 3 we see that the spring increases as the mass increase. Showing how the springs adapt. Iteration 2 shows opposite behavior that is quite strange. Although resulting in the Hinge-Damping value increasing in A.04.	 **A.06**	 **A.19**

Mass The weight of the vehicle. In A.20 we see the trainer finding the optimal mass for the vehicle (where only the mass value changes). Common behavior seems to be that a heavier vehicle is preferred.	 **A.07**	 **A.20**
Slider-Bounce How bouncy the spring system of the vehicle is. When training the slider group A.21-A.23 it is preferred that the bounce factor is at a minimum. This is true for all iterations except for iteration 1.	 **A.08**	 **A.21**
Slider-Damping The damping effect on the Slider-Spring. The preferred value for damping is one due to maximum correction when the vehicle is bouncing to much. When training group slider A.22 we see that most iterations the damping value is around 0.5.	 **A.09**	 **A.22**

Slider-Spring The amount of stiffness that the spring has against forces it bouncing against. Here we see that the slider (suspension) is increased to support the mass increase in A.06.	 **A.10**	 **A.23**
Suspension-Damping Wheels damping. Comparing graph A.24 to A.25 we see how the damping value and suspension value adapt to each other in iteration 6.	 **A.11**	 **A.24**
Suspension-String Wheels springiness. Seems that the preferred value for Suspension-spring is around 200.	 **A.12**	 **A.25**

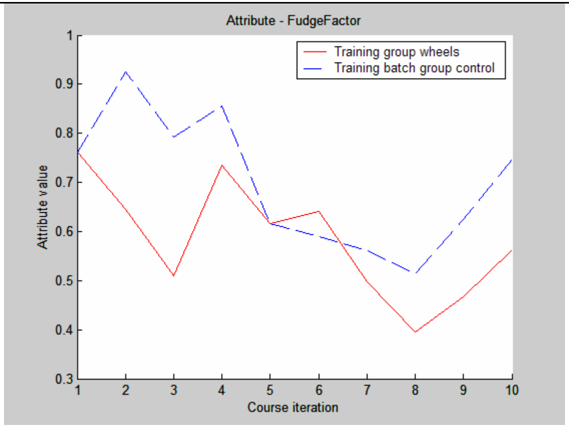
Suspension-StepSize The size of time step the physics simulation takes. High values give a slow motion effect. Starting of with the same value we see that this value scatters wildly across the scope.	 **A.13**	 **A.26**

Appendix B: "Evolution based on performance" charts



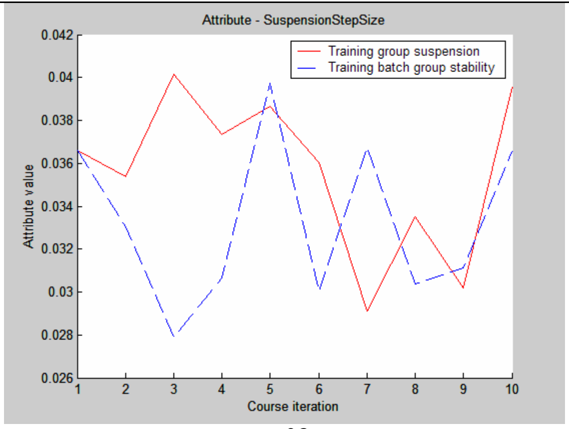
B.01

The power factor gets lower for each iteration. This may be due to more control when the vehicle is maneuvering heavily instead of high speeding a direct path.

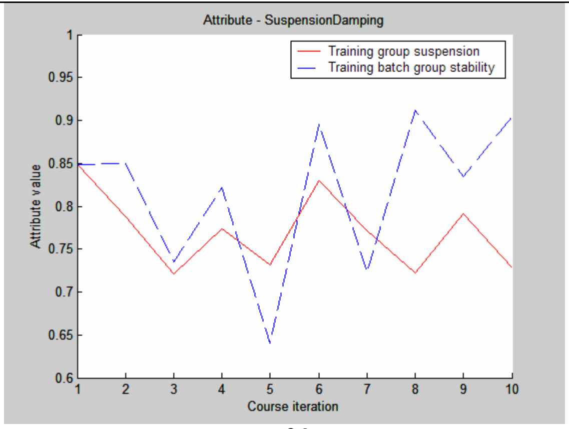


B.02

This may be for the same reason the power factor lowered. Due to more control when maneuvering.

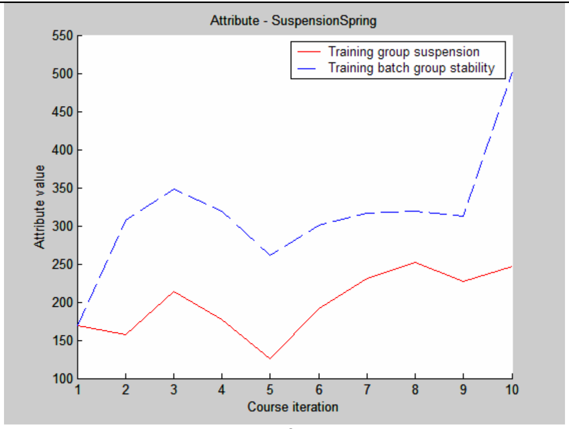


B.03



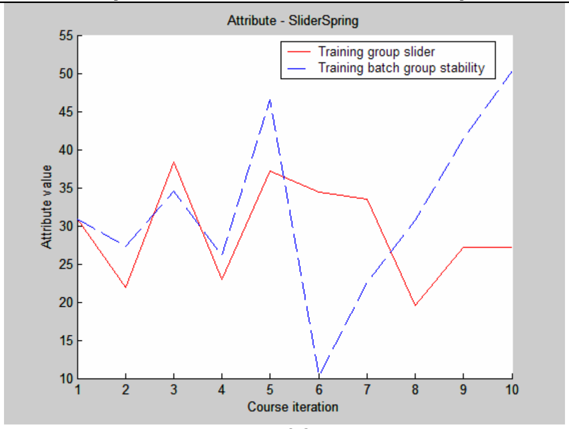
B.04

The blue doted line shows the group stability increase its damping value as B.05 increases its spring resulting in the wheels being harder. This happened due to the mass decreasing to reduce the vehicle from bouncing around.

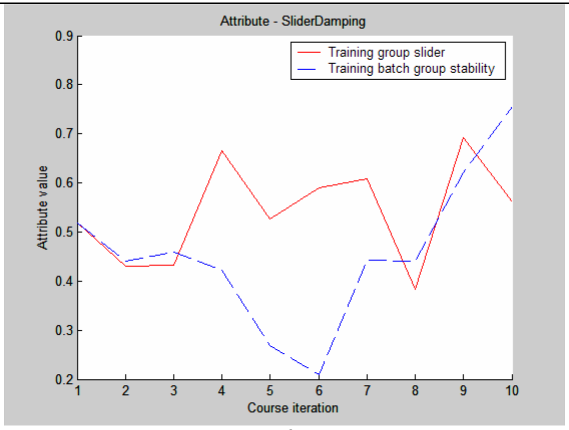


B.05

Here we see how slider spring B.06, slider damping B.07 and suspension string work together when training the group stability. To adapt a decreasing mass B.09

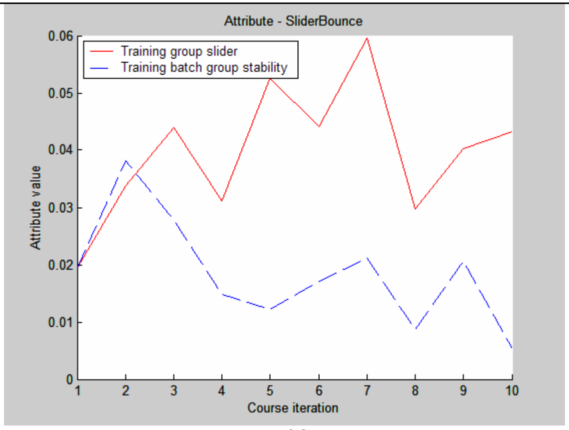


B.06

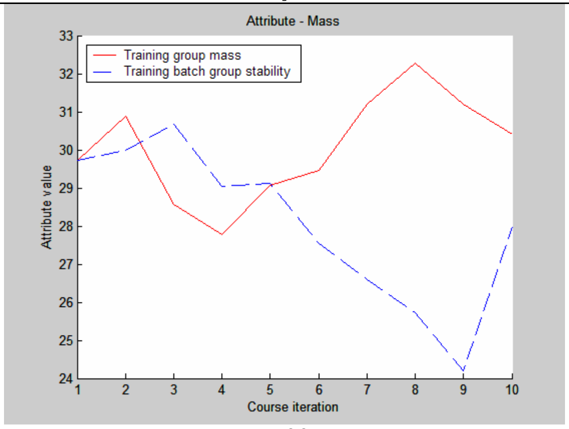


B.07

Here we see that when training group stability the trainer wanted more stability since both B.07 and B.08 converged to such states. This because mass decreased and made the vehicle lighter B.09.



B.08



B.09

The mass value went to extremes when training many attributes at once (stability) which is due to alterations of other attributes at the same time.
However when the mass is only trained we see that the trainer only diverged +2 to -2 which is promising?

The author first expected the two lines on this set of experiment to be closely related. Showing that an optimum values where found. This is how ever not the case since more values change simultaneously when training larger groups (as the stability group). Resulting in attributes adapting to changes done in the previous iteration.

Appendix C

```
from constants import *
from random import gauss
import time
import copy

class FORCarTrainer :

        car = None
        type = None

        #all available settings in a dictionary lookup as wheel as group type and their
stddev (sigma value)
        available_settings = {"FudgeFactor":["wheels",0.1], "PowerFactor":["wheels",0.1],

        "SteerFactor":["steer wheel",0.1], "SteerLimit":["steer wheel",0.1],
"SteerSpeed":["steer wheel",0.1], "SteerForce":["steer wheel",0.05],

        "BrakeFactor":["brakes",0.1],

        "HingeSpring":["hinge",50],"HingeDamping":["hinge",0.1],"HingeBounce":["hinge",0.0
1],

        "SliderSpring":["slider",7],"SliderDamping":["slider",0.1],"SliderBounce":["slider
",0.01],

        "Mass":["chassi",1.5]

        }

        available_parts = {}   #the structure that handles our gauss values
        train_parts = []               #the selected parts collection to be trained every
time we click for new value, ie 'wheel' or 'steer wheel' etc ...

        __strTableValues = ""

        def __init__ (self) :
        """ initalize all settings so we can work with them using gauss, generates a
        dictionary with inital values {"stddev":0.1, "mean":0.5, "temp":0} """
                for n in self.available_settings.keys() :
                         self.available_parts[n] = {"stddev":self.available_settings[n][1],
"mean":0.0, "temp":0}

        def logAttributeHeader (self) :
                self.__strTableValues = ""
                for a in self.train_parts :
                        self.__strTableValues = self.__strTableValues + str(a) + "\t"
                self.__strTableValues = self.__strTableValues + "\n"

        def logAttributeValues (self) :
                for a in self.train_parts :
                        self.__strTableValues = self.__strTableValues +
str(self.best_available_parts[a]["mean"]) + "\t"
                self.__strTableValues = self.__strTableValues + "\n"

        def flushLog (self) :
                f=open(PATH_LOG + "Race_" + str(time.time()) + ".log", 'w')
                f.write (self.__strTableValues)
                f.close()

        def setType (self, type) :
                """ accumilates the current """
                self.train_parts = []
                self.type = type
                for n in self.available_settings.keys() :
                        if self.available_settings[n][0] == type :
                                self.train_parts.append (n)
```

```
                              #print "training current type '" + n + "' from group '" + 
type + "'"
                              self.__restoreMeanValueForType (n)

       def setNewValue (self) :
              """ generates a new value for this collection of settings """
              for n in self.train_parts :
                     if self.available_parts.has_key (n) :
                            self.available_parts[n]["temp"] = 
gauss(self.available_parts[n]["mean"], self.available_parts[n]["stddev"])
                            self.__setNewValueForType (n, 
self.available_parts[n]["temp"])
                     else : print "no such key '" + n + "'"

       def setOldValue (self) :
              """ restores to previus value """
              for n in self.train_parts :
                     if self.available_parts.has_key (n) :
                            self.__setNewValueForType (n, 
self.available_parts[n]["mean"])
                     else : print "no such key '" + n + "'"

       def setNewValueToCurrent (self) :
              for n in self.train_parts :
                     if self.available_parts.has_key (n) :
                            self.available_parts[n]["mean"] = 
self.available_parts[n]["temp"]
                     else : print "no such key '" + n + "'"

       def __setNewValueForType (self, type, v) :
              """ calls the actual set function with parameter v using internal runtime 
evaluator """
              if self.available_parts.has_key (type) :
                     self.available_parts[type]["temp"] = v
              #print "Setting value for ", type, " to ", str(v)
              settingsGroupType = self.available_settings[type][0]
              if settingsGroupType == "steer wheel" :
                     eval ("self.car.getWheel(0).set"+type+"("+str(v)+")")
                     eval ("self.car.getWheel(1).set"+type+"("+str(v)+")")
              elif settingsGroupType == "wheels" :
                     eval ("self.car.getWheel(0).set"+type+"("+str(v)+")")
                     eval ("self.car.getWheel(1).set"+type+"("+str(v)+")")
                     eval ("self.car.getWheel(2).set"+type+"("+str(v)+")")
                     eval ("self.car.getWheel(3).set"+type+"("+str(v)+")")
              elif settingsGroupType == "slider" :
                     eval ("self.car.getAxis(0).set"+type+"("+str(v)+")")
                     eval ("self.car.getAxis(1).set"+type+"("+str(v)+")")
              elif settingsGroupType == "hinge" :
                     eval ("self.car.getAxis(0).set"+type+"("+str(v)+")")
                     eval ("self.car.getAxis(1).set"+type+"("+str(v)+")")
              else :
                     eval ("self.car.set"+type+"("+str(v)+")")

       def __getCarValueForType (self, type) :
              """ calls the actual get function setting the mean value using internal 
runtime evaluator """
              settingsGroupType = self.available_settings[type][0]
              if settingsGroupType == "steer wheel" :
                     return eval ("self.car.getWheel(0).get"+type+"()")
              elif settingsGroupType == "wheels" :
                     return eval ("self.car.getWheel(0).get"+type+"()")
              elif settingsGroupType == "slider" :
                     return eval ("self.car.getAxis(0).get"+type+"()")
              elif settingsGroupType == "hinge" :
                     return eval ("self.car.getAxis(0).get"+type+"()")
              else :
                     return eval ("self.car.get"+type+"()")

       def __restoreMeanValueForType (self, type) :
              """ calls the actual get function setting the mean value using internal 
runtime evaluator """
```

```
		self.available_parts[type]["mean"] = self.available_parts[type]["temp"] =
self.__getCarValueForType (type)

	def setCar (self, car) :
		self.car = car
		for n in self.train_parts :
			self.__restoreMeanValueForType(n)


class FORCarTrainerEvaluateRace (FORCarTrainer) :

	startTime = None
	endTime 	= None
	bestTime 	= None
	lapTime = None
	trainer 	= None #best setup
	evaluationCount = None
	best_available_parts = None
	isTraining = None

	def __init__ (self) :
		FORCarTrainer.__init__(self)
		self.startTime = 0
		self.endTime = 0
		self.bestTime = 1000000000000000
		self.evaluationCount = 0
		self.best_available_parts = {}

	def start (self) :
		""" creates new temp values for the vehicle v """
		self.startTime = time.time()
		self.evaluationCount = self.evaluationCount + 1
		#print "This is lap nr ", str(self.evaluationCount)

	def getIsTraining (self) :
		return self.isTraining

	def end (self) :
		self.endTime = time.time()
		if self.evaluationCount > 3 :
			self.isTraining = false
			for a in self.best_available_parts :
				self.available_parts[a]["mean"] =
self.best_available_parts[a]["mean"]
			FORCarTrainer.setOldValue(self)

	def beginSession (self) :
		self.startTime = 0
		self.endTime = 0
		self.bestTime = 1000000000000000
		self.evaluationCount = 0
		for a in self.available_parts.keys() :
			self.best_available_parts[a] = copy.copy(self.available_parts[a])
		self.isTraining = true

	def evaluate (self) :
		self.lapTime = self.endTime - self.startTime
		if self.bestTime > (self.endTime - self.startTime) : #got a better result
save settings
			self.bestTime = self.endTime - self.startTime
			#save the best values if we got them
			for a in self.available_parts.keys() :
				self.best_available_parts[a]["mean"] =
self.available_parts[a]["temp"]
		FORCarTrainer.setNewValue (self)
```