MorphEngine v1.0
Documentation Package

"Life Evolved on Earth after Seven Billion Years.
Do it in Seven Minutes."

Josh Bongard (bongard@ifi.unizh.ch)

May 21, 1999

Contents

1 Introduction **2**

2 Experiment I: Introducing The Hatstand **3**

3 Experiment II: Evolving The Hatstand **6**

4 Experiment III: Modifying The Hatstand **10**

5 Experiment IV: The Salamander and the Quadruped **17**

1 Introduction

The natural world is overflowing with examples of Nature's power and ingenuity: the echolocation strategies of bats; the carnivorous Venus fly-trap plant; human intelligence; the list is endless. However, Nature is as ingenious as it is slow. All Nature's creations are the result of evolution, a process that is measured in hundreds of millions of years. Evolution requires two important ingredients in order to function: *genetic variability* and *natural selection*. Genetic variability means that in a population of organisms, some organisms are better than others at surviving in their environment, and that some of these abilities are genetically determined. The ability of an organism to survive in its surroundings is referred to as its *fitness*. Natural selection means that organisms with low fitness will tend to die before they can reproduce, compared with organisms with higher fitness. In this way, higher fit organisms will produce more offspring, and as time passes, the population will contain increasingly more fit organisms. Over millions of years, this leads to a wondrous variety of organisms equipped with ingenious tools and weapons to grapple with their environment.

MorphEngine is an educational software package that allows the user to explore the workings of evolution on a PC, and see the results in a matter of minutes. The user first specifies the body of a creature, and what that creature should do. MorphEngine then designs a brain for your creature, known as a *neural network*, and then uses an algorithm based on evolution, known as a *genetic algorithm*, or GA, to refine the creature's brain so that it performs the desired task. The user can then view the behaviour of the evolved agent, as it moves about in a physically-realistic, three-dimensional virtual environment.

MorphEngine is designed such that the user is gradually introduced to various concepts from the field of artificial intelligence and artificial life, and allows the user to gain increasing control over these techniques in a fun and challenging environment.

The documentation is ordered into a set of increasingly sophisticated experiments. By conducting each experiment in turn, the user is introduced to more of the functionality of MorphEngine. Text describing how to conduct each experiment is written in **bold**. Descriptions of what the program is doing are interspersed among this text. Each experiment is followed by a series of questions and answers that provide additional information. It is not important to perform every experiment: no experiment follows directly from the results of the previous one. Some of the experiments contain follow-up questions for the advanced student.

2 Experiment I: Introducing The Hatstand

- Description: *In this experiment, a pre-defined creature, called The Hatstand, is tested in MorphEngine.*
- Concepts Introduced: *Basic usage of MorphEngine, the scripting language.*
- **Open up a DOS prompt, and move into the MorphEngine directory. In this directory, open the text file called** `hatstand.dat`**.** Scrolling through this file, you will notice a series of defined objects, enclosed in opening and closing brackets:

```
8 7
( LeftLeg
-position -0.5 0.1 0
-rotation 0 90 0
-shape    -cylinder 0.1 1.0
-mass     1
-colour   0.1 0.4 0.4
-floorContact
-addTouchSensor
)
...
```

  This file specifies the body plan of the hatstand: how many objects it is composed of, and how those objects are connected together. It also specifies what we want to the creature to do. For this exercise however, we won't go into the details of this file. Close the file now, and return to the MorphEngine DOS prompt.

- Now that we have a data file which specifies the body plan and the intended behaviour of our creature, MorphEngine can start designing a suitable brain, or neural network, for our creature. The neural network will coordinate the actions of our creature such that it performs the desired task, which is to move forward as fast as possible. **At the DOS prompt, type** `MorphEngine -l hatstand.dat`**.** This tells MorphEngine to begin evolving the brain for the creature specified in `hatstand.dat`. You can stop MorphEngine at any time by pressing `ESC`. If you are using a computer without a graphics card, visualization can be sped up by switching to wireframe by pressing `F2`. You can switch back to surface mode by pressing `F2` again. Reducing the size of the visualization window will speed up the simulation further. The hatstand creature is shown in Figure 1.
- *"What's happening?"* A window now appears showing the creature acting in its environment. All creatures in MorphEngine are rendered, evolved and analyzed in a physics-based simulator developed by MathEngine (`www.mathengine.com`

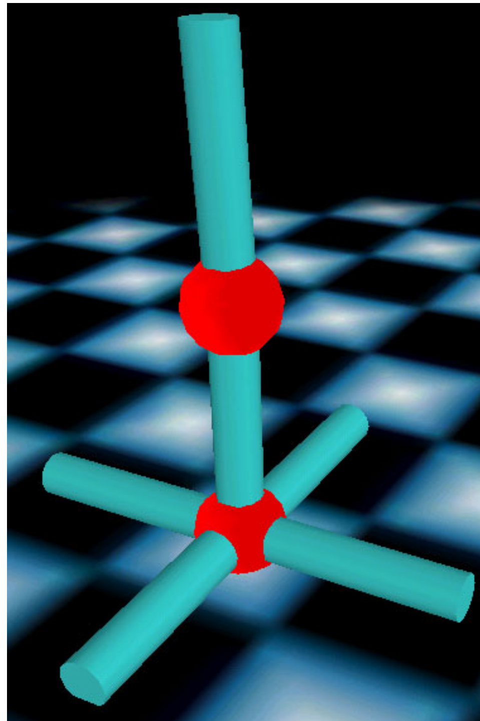

Figure 1: **The hatstand creature.**

). The MathEngine simulator provides a physically realistic, three-dimensional environment. MorphEngine constructs the user's creature in this environment, at which point the creature becomes subject to the laws of physics, such as gravity, friction, inertia, etc.

- *"The creature seems to be moving randomly, and only a little bit. Why?"* At first, MorphEngine generates creatures with random neural networks. That is, the creature's brain randomly takes signals from its sensors, and passes them to its motors. This usually leads to random motion, and because the motions are not coordinated, the creatures does not move very much.

- *"What's happening in the DOS window?"* The DOS window displays information about the evolutionary process. There is a command in `hatstand.dat` that tells MorphEngine to generate 10 random neural networks, and try each one out, sequentially, on the creature. When MorphEngine terminates (this can take between 5 and 20 minutes, depending on the speed of the PC), you should see something like the following in the DOS window:

```
Generation 0 of 1: Genome 0: [e: 1 f: -0.2056 l: 54]
```

```
Generation 0 of 1: Genome 1: [e: 1 f: -0.2869 l: 54]
Generation 0 of 1: Genome 2: [e: 1 f: -0.2417 l: 54]
Generation 0 of 1: Genome 3: [e: 1 f: 0.1841 l: 54]
Generation 0 of 1: Genome 4: [e: 1 f: 0.2459 l: 54]
Generation 0 of 1: Genome 5: [e: 1 f: 0.3081 l: 54]
Generation 0 of 1: Genome 6: [e: 1 f: -0.0968 l: 54]
Generation 0 of 1: Genome 7: [e: 1 f: -0.0267 l: 54]
Generation 0 of 1: Genome 8: [e: 1 f: -0.2610 l: 54]
Generation 0 of 1: Genome 9: [e: 1 f: 0.3884 l: 54]
Genome 9: [e: 2 f: 0.3884 l: 54]
```

In this experiment, MorphEngine has generated 10 random neural network blueprints, called genomes. The genomes are strings of numbers which describe a neural network: in the case of the hatstand creature, there are 54 numbers. The amount of numbers, or parameters, necessary to specify the brain of the creature is computed automatically by MorphEngine. The results of applying each of the random neural networks to the hatstand creature are displayed, each on a separate line.

No real evolution has taken place during this experiment: the 10 random neural network blueprints were generated, and MorphEngine tried out each, sequentially, on the hatstand creature. The `f:` parameter indicates how well the creature did with that particular neural network: `f:` stands for “fitness”. Because the task for this experiment was to move forward as fast as possible, the fitness indicates how far the creature moved forward, in a limited period of time. In the above example, we can see that the tenth neural network allowed the creature to move 0.3884 meters[1] forward during the allotted time period.

The last line of the output gives information about the best neural network so far. In this example, the tenth blueprint, genome 9, did the best.

[1] All lengths in MorphEngine are in meters; all weights are in kilograms.

3 Experiment II: Evolving The Hatstand

- Description: *In this experiment, the Hatstand creature is evolved to move forward as fast as possible.*
- Concepts Introduced: *Evolving creatures and analyzing the results.*
- Instead of just watching MorphEngine try out random neural networks, let's really let it try to evolve something interesting. In order to do this, we need to tell MorphEngine how evolution should happen. This is done in the creature's data file, in this case `hatstand.dat`.
- **Open** `hatstand.dat`**, and scroll to the bottom of the file.** There you will find information about how MorphEngine should evolve the creature:

```
(
-populationSize   10
-generations      1
-evaluationPeriod 300
-objectToMove     Base
)
```

-populationSize This parameter tells MorphEngine how many random genomes it should generate. This is known as the population. As evolution proceeds, genomes that lead to creatures with poor behaviour—low fitness—are removed from the population, and are replaced by copies of genomes that have produced creatures with good behaviour, or high fitness. MorphEngine tends to produce better results when working with large populations, but the price is that it takes much longer for the program to run. This is one of the many trade-offs that must be considered when working with artificial evolution.

-generations This parameter tells MorphEngine how many generations it should evaluate before terminating. MorphEngine uses a *genetic algorithm*, or GA, to simulate evolution: GAs are a biologically-inspired machine learning technique that automatically generates solutions to a problem. In this case, the "problem" is how to coordinate the actions of an agent to get it to do something interesting. The "solutions" are the neural networks that coordinate the actions by transforming signals from the agent's sensors into commands that are sent to the motors. More information about GAs can be found in "An Introduction to Genetic Algorithms" by Melanie Michell, or in "Genetic Algorithms + Data Structures = Evolution Programs" by Zbigniew Michalewicz. Figure 2 gives an overview of how GAs operate: the number of generations indicates how many times the GA should execute its main loop.

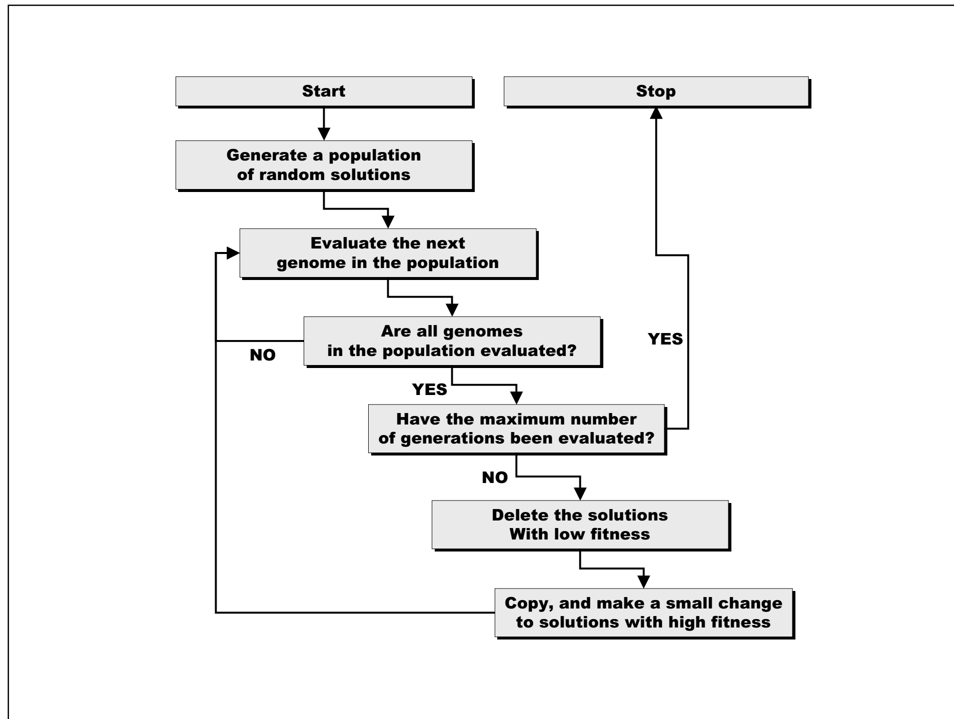

Figure 2: **An overview of the standard genetic algorithm**

-evaluationPeriod In order to determine the fitness of a particular creature, that creature must be evaluated in the virtual environment. This parameter indicates how long each evaluation period should last. A longer evaluation period allows the GA to better evaluate the fitness of the current creature, but the total time taken for the GA to execute is increased.

-objectToMove This parameter tells the GA which part of the creature's body will be used to determine the fitness. In other words, the fitness of a creature is given as the distance travelled by this body part in a forward direction.

- **Increase the population size to 50, and the generations to 2. Save** `hatstand.dat`**, and re-run morphEngine using** `morphEngine -l hatstand.dat -null`**.** The parameter `-null` tells the software not to visualize the behaviour of each creature. This allows MorphEngine to evaluate each solution much faster: the creature can be evaluated faster than real-time. When the renderer is turned off, you can no longer halt program execution by pressing `ESC`. You can now halt program execution at any time by pressing `CTRL-C`.

- MorphEngine will now generate a population of 50 random genomes, and will test each one on the hatstand creature. When all 50 creatures have been eval-

uated, and assigned a fitness value, the population is ranked according to fitness, and the worst 25 genomes are deleted. MorphEngine then probabilistically selects some of the remaining genomes, and copies them into the 25 remaining slots. When a genome is copied, a few small errors are introduced into the genome, known as *mutations*. The process is continued until the 25 slots have all been filled. The new genomes are then evaluated, and their fitness is determined. The population is then re-sorted according to fitness, and the program terminates.

- *"How can I see what happened?"* MorphEngine always stores the results of its execution in the `data` directory. **Go into the `data` directory, and display the contents using `dir`.** You should see something like:

```
RUN0_FIT DAT            44  04-19-01 10:54a run0_fit.dat
RUN0_G~1 DAT         2,067  04-19-01 10:54a run0_gen0.dat
RUN0_G~2 DAT           307  04-19-01 10:54a run0_ga16.dat
RUN0_G~3 DAT         2,046  04-19-01 10:55a run0_gen1.dat
RUN0_G~4 DAT           304  04-19-01 10:55a run0_ga51.dat
```

The `run0_gen*.dat` files provide information about each generation. **Open up `run0_gen0.dat` in an editor.** You should see something like:

```
-------------------------------
Generation: 0

Average fitness: -0.00278155
Best fitness: 0.0453808
-------------------------------
Genome 16: [e: 1 f: 0.0453808 l: 54]
Genome 2: [e: 1 f: 0.0452019 l: 54]
Genome 7: [e: 1 f: 0.0434307 l: 54]
Genome 35: [e: 1 f: 0.0371321 l: 54]
. . .
```

This file gives the current generation, the average fitness of population, and the fitness of the best genome so far, which in the example above was genome 16. (Each genome generated by MorphEngine is assigned an ID number, which in this case is 16.) MorphEngine always saves the most fit genome in the population into the `data` directory. In this example, the saved file was `run0_ga16.dat`.

- **Exit the editor, and now open `run0_gen1.dat`.** You should see something like:

```
-------------------------------
Generation: 1

Average fitness: 0.0227275
```

```
Best fitness: 0.0635221
------------------------------
Genome 51: [e: 1 f: 0.0635221 l: 54]
Genome 57: [e: 1 f: 0.0454222 l: 54]
Genome 16: [e: 1 f: 0.0453808 l: 54]
Genome 2: [e: 1 f: 0.0452019 l: 54]
```

In this example, two new genomes, genome 51 and 57, have allowed the creature to move slightly further forward than the previous best genome, genome 16. Let's have a look at this solution now.

- **Close the editor, and return to the** `MorphEngine` **directory. To analyze a particular solution, we supply the file name in which it is stored to MorphEngine. For the above example, this is done by typing** `MorphEngine -l hatstand.dat -a run0_ga51.dat`**.** Instead of evolving a moving creature, MorphEngine now repeatedly demonstrates the single solution stored in the data file. That is, the neural network specified in `run0_ga51.dat` is applied to the hatstand creature, and the resulting behaviour is continually replayed. Press `ESC` to terminate the program. When you are finished with a particular evolutionary run, it's a good idea to clear out the `data` directory, so that old data files are not confused with newer ones. **Enter the** `data` **directory, and remove all the data files by typing** `del *.dat`**.**

- *"The creature is still not moving much. Why?"* Remember, MorphEngine evaluated a total of only $50 + 25 = 75$ genomes, over 2 generations, to arrive at the final solution. Natural evolution has been working for hundreds of millions of years, acting on countless trillions of organisms: like natural evolution, the power of genetic algorithms is scalable. The larger the population size, and the more generations used, the better is the final result. **Try re-running the experiment, but increase both the population size and the number of generations.** MorphEngine will now take longer to run, but the output in the DOS window will give you an idea of how far along the program is. Remember, you can stop program execution when the renderer is turned off by pressing `CTRL-C`.

4 Experiment III: Modifying The Hatstand

- Description: *In this experiment, the hatstand creature is modified, and the impact on evolution is evaluated.*
- Concepts Introduced: *Using the script language.*
- Let's remove the top part of the hatstand creature and see what impact that has on the evolutionary process. In order to do this, we must remove the object from the script language called `UpperPole`, and also the *joint* which connects the upper pole to the rest of the creature. The first line of any creature data file indicates the number of objects composing the creature, and the number of joints which hold the objects together. **Open `hatstand.dat`, and save it as `hatstand2.dat`. Now, in the new creature data file, replace `8 7` by `7 6`.** This tells MorphEngine that the hatstand creature is now composed of seven objects, and six joints hold those objects together.

  Find the `UpperPole` object. Remove the object by deleting

```
( UpperPole
-position 0 1.6 0
-shape -cylinder 0.1 1.0
-rotation 90 0 0
-mass 1
-colour   0.1 0.4 0.4
)
```

  from the file. Now let's remove the joint which connects the upper pole to the upper sphere. **Find the `JointUpperPoleMidpoint` object, and delete**

```
( JointUpperPoleMidpoint
-connect       UpperPole Midpoint
-jointType     Hinge
-jointPosition Midpoint
-jointNormal   1 0 0
-jointLimits   -90 90
-addMotor
-addSensor
)
```

  from the file. Save the file, and return to the DOS prompt. Run MorphEngine with the renderer turned on, to see how the creature's morphology has changed, by typing `MorphEngine -l hatstand2.dat`. The modified hatstand creature should look like the creature shown in Figure 3.

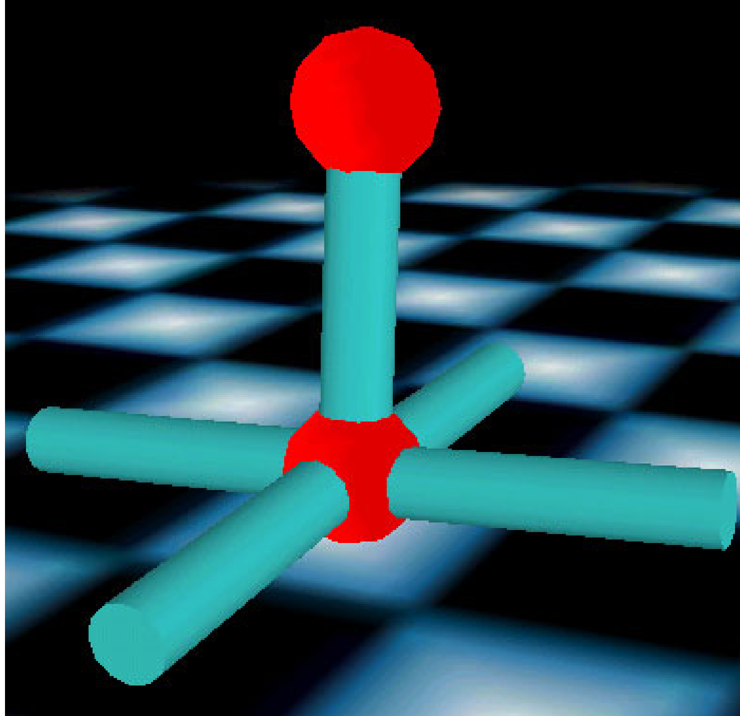

Figure 3: **The modified hatstand creature.**

- **Stop program execution, and re-run the program with the renderer turned off.**
- **After the program terminates, view the creature from the last generation.** This can be done in either of two ways.
  - **Enter the** `data` **directory, and view the last generation file.** If MorphEngine ran for 10 generations, open `run0_gen9.dat` (generations are counted starting from 0, so the tenth generation is generation 9). The first genome in the genome list indicates the most fit genome from the final generation. **Note the ID of the most fit genome, and return to the DOS prompt. Now re-run MorphEngine, and analyze this genome.**
  - **Enter the** `data` **directory, and list all of the genome files by typing** `dir *ga*`**. Note the genome file with the highest ID number, return to the** `MorphEngine` **directory, and supply this file to MorphEngine.** After each generation, MorphEngine saves the data file associated with the most fit genome in the population. As new genomes are generated, they are assigned new ID numbers. Thus, the genome file with the highest ID number was the most recently saved data file, and the current most fit genome

produced by the evolutionary run.

- *"When I use the new creature body plan, the number of parameters in the genome has been reduced from* 54 *to* 47*. Why?"* This has to do with how MorphEngine creates a brain for your creature, based on the body. Every creature in MorphEngine must contain at least one *sensor*, and at least one *motor*. Sensors allow the creature to sense the state of the world around them; motors allow the creature to move its body, and perform actions in the world. The neural network transforms sensory signals into motor commands, thus allowing the creature to perform particular actions based on particular sensory stimulus. When the creature moves, its picture of the outside world—as seen through its sensors—changes. The new sensor readings may lead to different motor commands, which produce different actions, and the sensor signals will again change. By fine-tuning the transformation from sensor signals to motor commands based on the given task, the creature begins to exhibit increasingly better behaviour. In his book "Vehicles", Valentino Braitenberg showed how very simple tranformations can lead to seemingly intelligent behaviour.

  For more information about neural networks applied to agents and robots in this way, refer to "Understanding Intelligence" by Rolf Pfeifer and Christian Scheier. For more information on neural networks in general, refer to:

  - The online neural network script offered by the University of Zürich at `http://www.ifi.unizh.ch/ailab/teaching/NN2001/`,
  - Neural Nets by Kevin Gurney at `http://www.shef.ac.uk/psychology/gurney/notes`,
  - The Neural Net FAQ at `http://www.informatik.uni-freiburg.de/ heinz/FAQ.html`, or
  - "Neural Networks" by Simon Haykin.

  In the original hatstand data file, notice that several of the objects and joints contain sensors and motors: the four legs, and the base sphere, contain *touch sensors*; the joint connecting the upper pole to the midpoint sphere contains an *angle sensor*, referred to as a *proprioceptive sensor*; the four joints connecting the legs to the base sphere contain *motors*; and the two joints connecting the upper pole to the midpoint sphere, and the lower pole to the base sphere contain motors.

  Currently, there is one type of motor which can be implanted in a MorphEngine creature, and three types of sensors.

  Motor: Motors in MorphEngine can be attached to a creature's joints. At the moment, motors can only be applied to hinge joints: that is, joints that connect two objects together, and rotate them relative to each other through a two-dimensional plane. The human elbow is an example of such a joint. Motors take as input an angle, and apply the necessary force to the joint

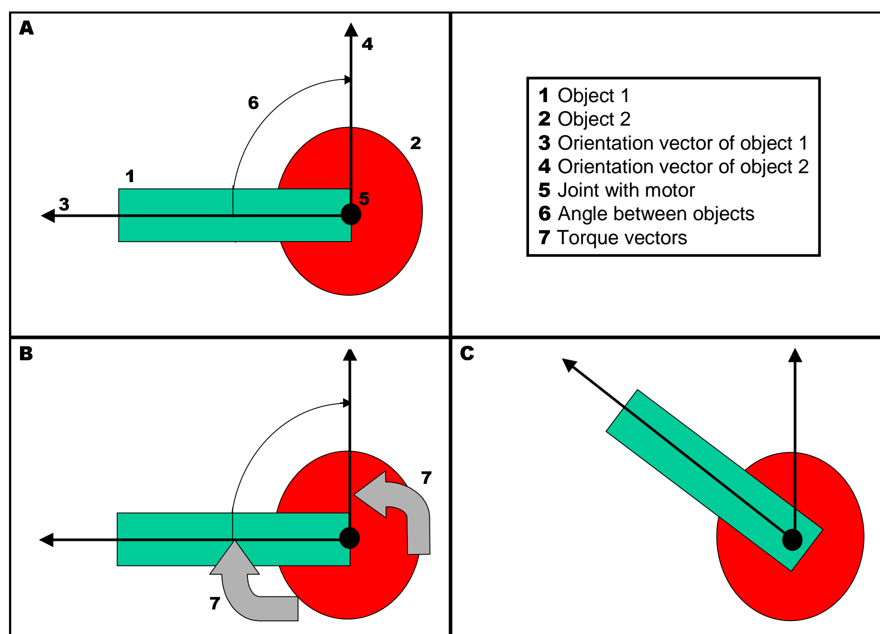

Figure 4: **A motorized joint.** In panel A, the two objects are connected by a motorized joint. Each object has an orientation, which can be described by a three-dimensional vector. The angle between these vectors is referred to as the object pair's *relative orientation*. In panel B, the motor is supplied with a low angle; rotational force, or *torque*, is then applied to both objects to reduce their relative orientation. The final relative orientation between the objects is shown in panel C.

such that the two objects rotate relative to each other until the angle between them matches the input signal. Figure 4 illustrates one such action.

Touch Sensor: This sensor returns a positive signal when the object in which it is embedded is in contact with the ground. Otherwise, it returns a negative signal.

Proprioceptive Sensor: This sensor can be placed in joints, and returns a signal commensurate with the angle between the objects connected by the joint: if the angle is small, a negative signal is returned; if the angle is large, a positive signal is returned.

Light Sensor: This sensor can be embedded in objects, and returns a signal commensurate with the distance between that object and an external light source: if the object is close to the light, a negative signal is returned; if the object is far away, a positive signal is returned.

After MorphEngine has determined how many sensors and motors are to be included in a given creature, the software "wires up" the sensors to the motors. Just how the sensors and motors are wired up is based on the numbers of sensors and motors placed into the creature. Figure 5 shows the neural net-

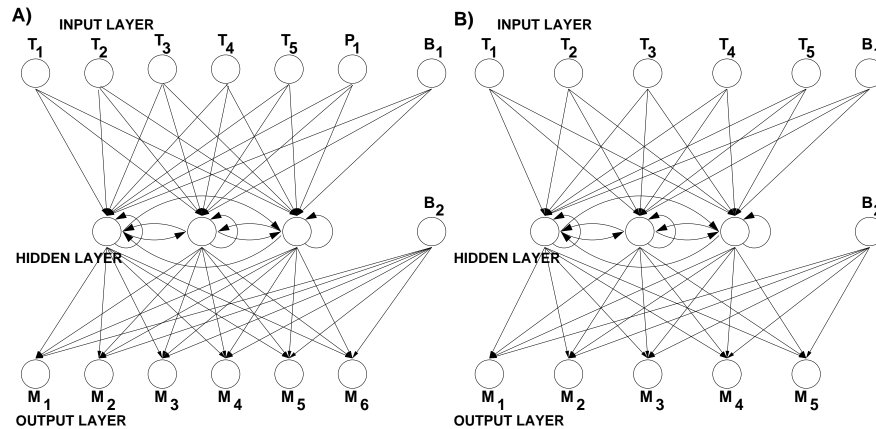

Figure 5: **Neural network topology for the hatstand creatures.** Panel A shows the neural network for the original hatstand creature: the signals from five touch sensors and one proprioceptive sensor are translated into six motor commands, using 54 connections, or synapses. Panel B shows the neural network topology for the modified hatstand creature. Here, the signals from only five touch sensors are translated into five motor commands, using 47 synapses.

---

works that are generated for the original, and the modified hatstand creature. Note that although the `UpperPole` object did not contain any sensors, the `JointUpperPoleMidpoint` joint that was deleted contained both a proprioceptive sensor and a motor. Thus, in the modified hatstand creature, less "wires", or synapses are needed to connect the sensors to the motors.

- *"What is the hidden layer shown in Figure 5?"* The hidden layer is an additional layer of neurons, called hidden neurons, that generalizes the transformations possible from sensory signals to motor commands. Without a hidden layer, only linear transformations are possible. With a hidden layer, non-linear transformations are possible.

- *"The hidden neurons seemed to be connected to one another, while the sensor and motor neurons aren't. Why?"* By including recurrent synapses in the hidden layer, it is possible for the creature's brain to use *memory*: the values stored in the hidden neurons are dependent on the incoming signals from the input layer, but also the values stored in the hidden layer from the previous time step. Whether the creature will actually utilize memory when performing its behaviour depends on the specific task, and what solution the genetic algorithm is pursuing.

- *"How does the genetic algorithm optimize a creature's neural network to perform the chosen task?"* The genomes in the genetic algorithm population store the strengths, or *weights* of the synapses for the creature's neural network. Synapse weights are floating-point values, and can be either positive or negative. If two neurons are connected by a synapse with a high positive weight, then the values of the originating and the target neuron will be similar. If they are connected by a synapse with a high negative value, the value of the target neuron will be

inversely correlated with the value of the originating neuron. If the weight is near zero, there will be little correlation between the values of the two neurons. The values of the sensor neurons are provided by the sensors. The values of the hidden neurons are computed by

$$h_j(t) = \sum_{k=1}^{I} w_{kj} i_k(t) + \sum_{l=1}^{H} w_{lj} h_l(t-1) + w_{bj}, \tag{1}$$

where $h_j(t)$ is the value of the jth hidden neuron at time step t, H and I are the number of hidden and input neurons respectively, w_{kj} is the weight of the synapse connecting the kth input neuron to the jth hidden neuron, $i_k(t)$ is the value of the kth input neuron at the current time step, w_{lj} is the weight of the synapse connecting the lth hidden neuron to the jth hidden neuron, $h_l(t-1)$ is the value of the lth hidden neuron from the previous time step, and w_{bj} is the synaptic weight between the bias neuron and the jth hidden neuron.

However, the values of the hidden neurons may now be much larger than 1, or much less than -1. The values of the neurons are smoothed by the function

$$h_j(t) = \frac{2}{1 + e^{-h_j(t)}} - 1, \tag{2}$$

so that they again range between -1 and 1.

The values of the output neurons are computed using

$$o_j(t) = \sum_{k=1}^{H} w_{kj} h_k(t) + w_{bj}, \tag{3}$$

where $o_j(t)$ is the value of the jth output neuron at time t, w_{kj} now represents the weight of the synapse connecting the kth hidden neuron to the jth output neuron, $h_k(t)$ is the value of the kth hidden neuron at time t, and w_{bj} is the weight of the synapse connecting the bias neuron to the jth output neuron. And again, the value of the output neuron is smoothed to the range $[-1, 1]$ by

$$o_j(t) = \frac{2}{1 + e^{-o_j(t)}} - 1, \tag{4}$$

The genetic algorithm optimizes the behaviour of the creature by modifying the set of synaptic weights for the neural network, and thus modifying the transformation of the sensor signals into motor commands. As evolution proceeds, the GA discovers better combinations of synaptic weights, and the creature's performance increases over time.

- It is possible to view the behaviour of the neural network of a creature as it is performing its behaviour, by supplying the `-n` flag when the program is started. **Return to the DOS prompt, and run the program by typing** `MorphEngine -l hatstand.dat -n`**.** MorphEngine now pauses after each time step of a creature's evaluation, and prints the values of the neural network in the DOS window, such as

```
1.00 1.00 1.00 1.00 -1.00 -0.37
-0.73 -0.02 0.96
0.28 0.73 0.12 -0.05 -0.26 -0.36 .
```

The first line prints out the values of the input neurons, which reflect the values of the creature's sensors: for the hatstand creature, this includes five touch sensors and one proprioceptive sensor. The second line prints out the values of the three hidden neurons; the third line prints out the values of the six output neurons.

To iterate through the simulation, focus the DOS window and press ENTER repeatedly. Re-run MorphEngine using the modified hatstand creature and the `-n` flag, and note the difference in the neural network.

- *"How are the output neuron values translated into motor commands?"* In `hatstand2.dat`, look at any of the four joints connecting the legs to the base object, `JointLeftLegBase`, `JointRightLegBase`, `JointFrontLegBase`, or `JointBackLegBase`. In these objects, note the line `-jointLimits -45 45`. This tells MorphEngine that this joint has a limit on how far it can rotate: it can rotate 45 degrees either way from its starting position. **Try flexing your elbows and knees: what are the joint limits on the elbow and knee joints?** There are four output neurons associated with the motors in these four joints, the values of which can range between -1 and 1. At the end of each time step, the values from these output neurons are scaled between -45 and 45, and input into the motors. In subsequent time steps, the motors apply torque to the two objects connected by the joint such that the angle between the objects approaches the input angle.

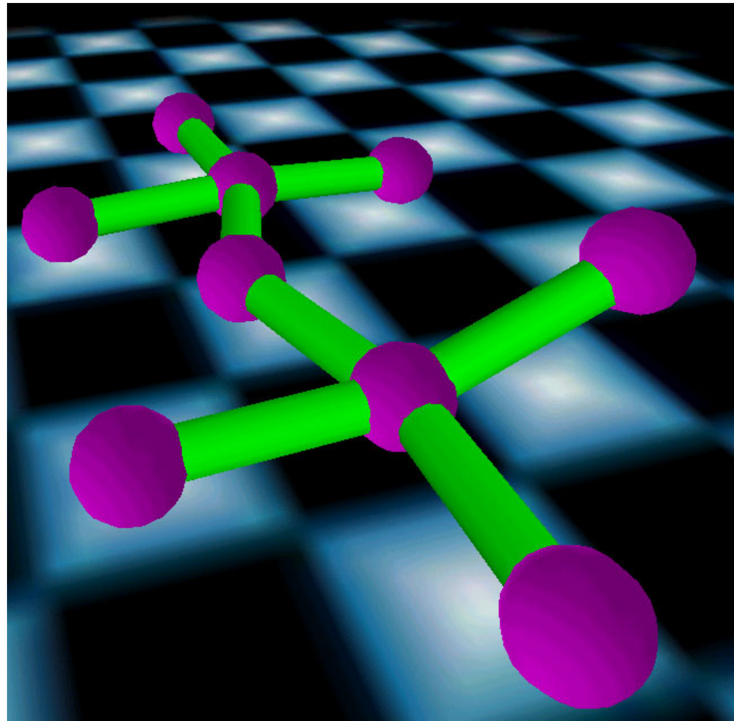

Figure 6: **The salamander.**

5 Experiment IV: The Salamander and the Quadruped

Let's turn now to a more interesting creature, the Salamander. This creature has a flexible spine, and four short limbs attached to the spine. Figure 6 shows the body plan of this creature.

- Description: *In this experiment, locomotion is evolved for the salamander. The salamander is then modified to create a quadruped body plan, and locomotion is again evolved for the new creature.*
- Concepts Introduced: *Detailed use of the script language.*
- **Open the** `salamander.dat` **salamander data file. Scroll to the bottom of the file.** Like the hatstand creature, the evolutionary task for the salamander is to move forward as quickly as possible. Here though, we tell evolution that it must maximize the forward distance of the `Socket1` object, which serves as the head of the salamander, and is the purple sphere further "into" the computer screen.

Close the `salamander.dat` **file.** Note that the salamander will be evolved using a population size of 100 genomes, and for 100 generations.

- **Evolve the salamander by typing** `MorphEngine -l salamander.dat -null` **at the DOS prompt. View the results at the end of evolution, or by stopping the program prematurely using** `CTRL-C`**.**

- The course that artificial evolution takes, like natural evolution, depends very much on the initial conditions: that is, the genetic information stored collectively in the evolving creatures. By starting with a different random set of initial solutions, artificial evolution very often comes up with different solutions to the same problem. Most of the time, solutions from one run tend to outperform the best solution found in another run. When experimenting with artificial evolution in general, and genetic algorithms in particular, it is a good idea to perform a number of evolutionary runs, starting with different random populations. Let's try this now. **Re-evolve locomotion for the salamander by typing** `MorphEngine -l salamander.dat -null -r 1` **at the DOS prompt. View the results of evolution by supplying one of the genome data files to MorphEngine.** Make sure to keep the genome files in the `data` directory this time, as you will need them for the next step of the experiment.

- *"What does the* `-r 1` *do?"* This command line parameter specifies the *random seed* that should be used by MorphEngine. The random seed is used to begin the generation of random sets of numbers. If the program is re-run, and the same random seed is supplied, the computer will generate the same random set of numbers. Because the random genomes are generated by stringing together random numbers, different random seeds supplied to MorphEngine produce different starting populations. If the `-r` switch—which stands for "random seed"—is not given, MorphEngine uses a random seed of 0. Any integer can be used as a random seed. **Let evolution finish, or stop it prematurely using** `CTRL-C`**. View the results of evolution by analyzing the most recent genome file saved in the** `data` **directory. Note that the genome files are now preceded by** `run1_` **instead of** `run0_`**.** The `run*` prefix in the data files indicates to the user which random seed was used to produce these data files.

- Does the evolved locomotion look any different from the one evolved using the default random seed? Does the new evolved locomotion pattern allow the salamander to get further than the previously evolved locomotion pattern? You may find that some of the final results of evolution often are much worse than other evolved solutions, just because they started with a particular random population. This is often known as *historical accident* in biology: evolution produces the solutions it does partially because it is constrained by the solutions that were evolved before.

- Let's now modify the body plan of the salamander. **Open the** `salamander.dat` **data file, and locate the** `JointLeftArmSocket2` **description:**

```
( JointLeftArmSocket2
```

```
-connect        LeftArm Socket2
-jointType      Hinge
-jointPosition Socket2
-jointNormal    0 0 1
-jointLimits    -45 45
-addMotor
)
```

This description specifies the joint connecting the left arm of the salamander to the second purple sphere behind the head, called `Socket2`. The joint is a `hinge`, or one degree-of-freedom rotational joint (see Figure 4.) The fulcrum of the joint, or the point about which the two objects connected by the joint rotate, is centered in `Socket2`. The `jointNormal` command specifies how the joint should rotate: joints always rotate within the plane that lies perpendicular to the three-dimensional vector joint normal. Because this joint has a joint normal that lies along the z axis (in MorphEngine, the x axis ranges from left to right; the y axis stretches up and down; and the z axis points into and out of the screen), the left arm will rotate within the plane that contains the x and y axes. **Run one of the evolved solutions for the salamander. Press `F8` to pause the simulation. Zoom in on the left arm using the mouse: by left clicking and moving the mouse in the simulation window, the camera position changes; by right clicking and moving the mouse changes the camera target position; pressing $+$ and $-$ zooms in and out. Continue press `F8` to step through the simulation. Press `CTRL-F8` to continue the simulation.** Note how the left arm rotates about the spine of the salamander.

- **Change the normal of `JointLeftArmSocket2` from $0\ 0\ 1$ to $0\ 1\ 0$, and the joint limits from -45 and 45 to -90 and 90. Save the data file, and re-run evolution with the renderer turned on.** How has the behaviour of the salamander changed? How is the left arm rotating relative to the spine?

- **Change the normal of `JointLeftArmSocket2` from $0\ 1\ 0$ to $1\ 0\ 0$. Save the data file, and re-run evolution with the renderer turned on.** How has the behaviour of the salamander changed? How is the left arm rotating relative to the spine?

- **Change the joint normal back to $0\ 0\ 1$, and the joint limits back to -45 and 45 degrees.**

- It should now be clear that the `jointLimits` parameters indicate the minimum and maximum rotational limits of the joint. Let's now modify the morphology of the salamander more drastically. **Change the limits for `JointLeftArmSocket2` and `JointLeftLegSocket4` from -45 and 45 to 45 and 90, and the limits for `JointRightArmSocket2` and `JointRightLegSocket4` from -45 and 45 to -90 and -45. Save the new body plan file as `quadruped.dat`. Re-run evolution, with the renderer turned on.** We've now changed the salamander from a creature that moves by sliding along the ground into a quadruped,

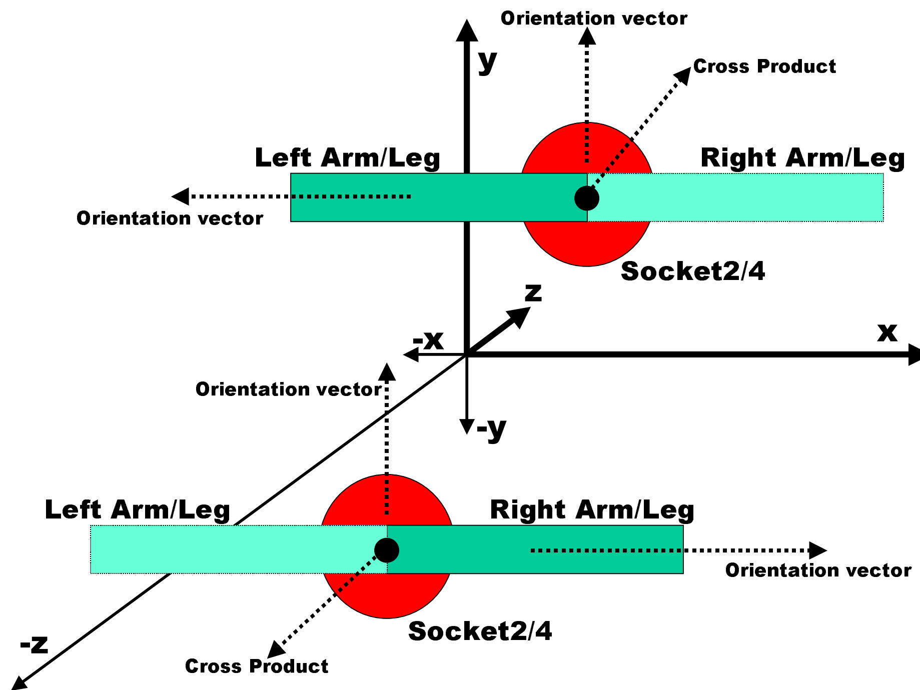

Figure 7: **Joint limit computation.** The cross products for the left arm and leg and socket 2 and 4 point along the negative z axis. The cross products for the right arm and leg and socket 2 and 4 point along the positive z axis.

---

or a creature that moves by walking on four legs. **Evolve a neural network for the quadruped creature, with the renderer turned off. Try several evolutionary runs with different random seeds.** What kind of locomotion patterns does this creature use? Are they similar to the salamander gaits you've already evolved? How do they differ?

- *"Why do the joint limits on the left side of the quadruped differ from those on the right side?"* Each object in MorphEngine has a three-dimensional position, and three-dimensional orientation. Both parameters are specified by a three-dimensional vector. The orientation vectors for the left arm and leg point left, along the negative x axis; the orientation vectors for the right arm and leg point right, along the positive x axis. The orientation vectors for `Segment2` and `Segment4` point upwards. The joint angles are computed based on the *cross product* of the orientation vectors of the two objects connected by the joint. For this reason, the vectors formed by these cross products are equal but opposite in sign between the joints on the left side of the body, and the right side of the body. Figure 7 gives a graphical depiction of joint limit computation in MorphEngine.

- **Try changing the limits of other hinges in the salamander and quadruped, and gauge what effects, if any, this has on the typical evolved locomotion patterns.** It is possible to remove a joint altogether by turning it into a `fixed` joint. Remove the joint normal, joint limits, and the motor, if necessary, and change the `hinge` joint specification to `fixed`. Conversely, fixed joints can be changed into hinge joints. **Change some of the hinge joints into fixed joints, and some of the fixed joints into hinge joints. See how this affects the evolved behaviour of the creature.**

- *"What are the parameters that specify objects in MorphEngine?"* As mentioned above, each object in MorphEngine has both a position and orientation. These parameters of an object are set using the `-position` and `orientation` parameters. For example, the head of the salamander, called `Socket1`, starts at position 0 0.2 2. **Try changing the position of the head, save the data file, and read it into MorphEngine. View the results.**

- Because the head is a sphere, its orientation does not need to be specified. However the legs and arms, which are composed of cylinders, lie, by default, along the z axis. They must be rotated to lie perpendicular to the salamander's spine, that is, along the x axis. In order to accomplish this, they are rotated about the y axis by 90 degrees: they are rotated 90 degrees through the x-z plane. Note that the `LeftArm`, `RightArm`, `LeftLeg` and `RightLeg` objects contain the line `-rotation 0 90 0`. **Try changing the orientations of the salamander's arms and legs, and view the results.**

- Besides the position and orientation of objects, their sizes need also be specified. The command `-shape -sphere 0.2` given for each of the spine sockets specifies that these objects are spheres, and have a radius of 0.2 meters, or 20 centimeters. The command `-shape -cylinder 0.1 1` given for the arms and legs indicate that these objects are cylinders, and they have a radius of 0.1 meters, and a length of 1 meter. `Try changing the sizes of the objects composing the salamander, and view the results.`

- Additional parameters for the objects in MorphEngine give the masses and colours of these objects. The parameter `floorContact` included in the hands, feet and spine sockets of the salamander indicates that these objects can come in contact with the floor, and should not pass through it. The fewer $floorContact$ objects that MorphEngine needs to calculate, the faster each evaluation will take, and thus the faster evolution can proceed. **Try removing the `floorContact` parameters from the spine sockets in the quadruped creature. View the results.** Because the quadruped has failed at moving forward properly if it falls over, we can terminate these evaluations early, and not wait until the evaluation time expires. We can do this by checking when one of the spine segments, say the middle spine segment, falls below a certain height. **Add the line `-terminateIf Socket3 0.1` to the evolution parameters at the bottom of the quadruped data file. Save the file, and re-run evolution.** The evolution parameters should now like like

```
(
-populationSize   100
-generations      100
-evaluationPeriod 200
-hiddenNodes      3
-objectToMove     Socket1
-terminateIf      Socket3 0.1
)
```

Note now that if the quadruped falls over, the current evaluation finishes, and the next evaluation begins. We have indicated to MorphEngine that if the middle socket, `Socket3`, falls to 0.1 meters above the ground, the quadruped has fallen over, and the next evaluation can begin.

- You now have sufficient knowledge to begin creating your own creatures, and evolving behaviour for them. The best place to begin is by removing and adding objects and joints to the existing creatures: remember always to back up the data file of the original creatures first. Remember when adding or deleting objects and joints, you must update the first two numbers on the first line of the data file: the first number indicates how many objects the creature is composed of, and the second number indicates how many joints connect those objects together. Have fun!

- Questions and comments should be directed to the author. Consult `www.ifi.unizh.ch/ailab/people/bongard` for future MorphEngine versions.