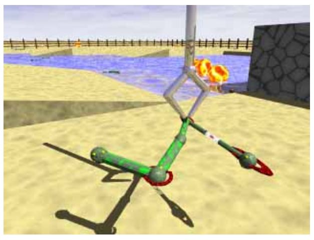
FRAMSTICKS MANUAL

This is a pdf form of the Framsticks documentation. It is compiled from the Framsticks HTML web pages which look better (and are more recent) than this document. It is strongly recommended to browse on-line documentation when possible. This reference manual, albeit outdated (created in 2008), lists selected issues, includes a Table of Contents, and may be more suitable for printing or serve as an off-line reference.

www.framsticks.com

by Maciej Komosinski and Szymon Ulatowski



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

You are welcome to try Framsticks! Users of this software work on evolutionary computation, artificial intelligence, neural networks, biology, robotics and simulation, cognitive science, neuroscience, medicine, philosophy, virtual reality, graphics, and art. The system can be interesting for experimenters who would like to evolve their own artificial creatures and see them in a three-dimensional, virtual world. You can also manually design and test creatures. This software is a versatile tool for research and education.

Table of Contents

Table of Contents

Table of Contents

1 Artificial Life introduction

The objective of this project is to study evolution process in a computer-simulated artificial world. We hope that, like in the real world, despite randomness and aimlessness of basic evolution mechanisms, it will lead to creation of more and more efficient artificial organisms, still better and better adapted to the artificial world conditions.

For many years people used computers to simulate Nature. This kind of research also belongs to a field of science called "Artificial Life" (AL). The name is obviously connected with "Artificial Intelligence" – both fields of study partly overlap, but AL has more in common with biology and physics. It might be called a branch of biology, because we study living (or "living", depending on the definition of this word) organisms in an environment. No matter that the environment is an artificial, virtual world inside a computer: philosophers haven't yet decided whether our world is real and, nevertheless, biologists keep examining its living organisms. In addition, our made-up worlds let us (until we visit other, very distant planets) study signs of life which has nothing to do with proteins.

Scientists studying AL concentrate on various fields and serve many purposes. Rules of simulated worlds do not have to be like real ones, but such models seem more interesting (perhaps because you are able to directly compare simulation results with reality).

Boids are an example of creatures following a few simple rules, which nevertheless make them group with others (like fish shoals or bird herds). This kind of behavior was used in film making (computer control of animals' models – "Lion King", "Batman forever" and many advertisements). More sophisticated models include learning and self-improving creatures. Artificial organisms (i.e. carriages with engines) can be taught to avoid obstacles, seek energy spots, follow targets, escape enemies etc. Such experiments are directly connected with real robots' control – they might alone perform useful actions.

Often it is evolution which controls improvements – it awards better-adapted, fit (or, in the case of artificial world, efficient as we want them) organisms. That has to do with genetic algorithms and works the same way. In Nature, organism morphology is determined by genes, and this – together with other mechanisms, like learning – supports evolution and its high efficiency. In our life simulator, similarly, genes describe the whole structure of an organism. Full freedom in building genotypes means theoretically an ability for creation of creatures of any complexity.

Artificial Life research makes us ask whether self-improvement of artificial organisms can lead to founding consciousness, intelligence, feelings? Not in our simple model, of course. However, there is still a question if evolution alone created organisms which live among us. Experiments with simulated evolutions will show their capabilities, and the results of such experimentation can help explain the mystery of our creation.

If you would like to share your ideas on these topics or read other comments, do not hesitate and use our on-line forums.

2 About the project – objectives and scope

2.1 Objectives and scope of this research

The objective of these experiments is a study of evolution capabilities of creatures in simplified Earth-like conditions. They are: a three-dimensional environment, genotype representation of organisms, physical structure (body) and neural network (brain) both described in genotype, stiumuli loop (environment – receptors – brain – effectors – environment), genotype reconfiguration operations (mutation, crossing over, repair), energetic requirements and balance, and specialization.

The simulator allows the study of both directed (with fitness criterion defined) and spontaneous (with no such criterion) evolution. In the directed case, it is possible to "grow" creatures with the given properties, like simple construction and smooth movement, strength and robustness, ability to move in land and water environments, seeking food, following targets, escaping and many others. The system allows users to create more experiment definitions, which may lead to unexpected results and emergence of very complex behaviors.

Another aspect of this research is the influence of representation (description) of creatures, reconfiguration operators and the rules of organism building on evolution results and characteristics of structures of created individuals.

The most important part of the research is the study and evaluation of capabilities of various evolutionary processes, including those concerning undirected evolution (which has not already been done in such a complex environmental and simulation conditions).

2.2 System capabilities

Framsticks has a powerful, universal simulator with great capabilities, which are:

- Three-dimensional, mechanical simulation of the artificial world (*Mechastick* module):
    - Creatures' simulation with finite elements method
    - Specialization of their limbs – friction, strength, ways of acquiring energy by assimilation, ingestion etc.
    - Flat land, terrain made of blocks/slopes of different height and water environment
    - Nondestructive and destructive collisions
    - Ability for the user to interact with the simulated world (dragging of creatures, energy balls placement, revival and killing of individuals)
- Simulation of creatures' control system ("brain"):
    - Neural net of any topology, built from any kind of neurons
    - Interaction with the environment: receptors (touch, equilibrium and energy localization senses) and effectors (muscles moving a creature, controlled by its neural network)
    - Various types of neurons
- Evolution:
    - Maintaining a set of genotypes grouped in gene pools
    - Maintaining a set of individuals grouped in populations
    - Many ways of organism description (genetic representations)
    - Modification of creatures' descriptions by crossover and mutations
    - Evaluation of creatures with various criteria (life span, speed etc.)

    - ♦ Maintaining of creatures' energetic balance (gains and usage of energy for various purposes)
    - ♦ Performance measurement, multi-criteria evaluation
- Scripting language which allows for modification and extension of the system, designing custom user experiment definitions, custom neuron types, user-defined fitness functions, macros, visualization styles, etc.
- Parametization of most of operations, environment, simulation and evolution rules, and system behavior
- Graphical interface for Windows and Amiga platforms, textual interface for any system

2.3 Directions of work

- various genotype encodings (including developmental) to describe body and brain
- network version of the simulator, and specification of the communication protocol
- JAVA and Python network interfaces independent from the simulator; other interfaces from Framsticks users
- more accurate physical simulation
- design of experiment definitions for open-ended, spontaneous evolution and biologically-plausible ecosystem simulation
- additional senses (receptors)
- network-distributed evolution; Framsticks Experimentation Center

- developers should refer to **Framsticks Development Center**.

3 Assumptions of the model

We tried to make the artificial world similar to the real one. When creating it, we wanted to support it with all the features which allow the evolution process to be aimed at a direction, so that organisms can – without a purpose – discover new ways of living according to their fitness defined by the user. On the other hand, we tried to create reasonable conditions of living in the environment without any fitness criteria defined – that is, in a spontaneous evolution.

Biological evolution started from simple components. Much time had passed until the first creatures were able to reproduce. In our artificial world we skip this "chemical evolution" stage. We supply our creatures with basic functions: notation of their features in their genotypes, multiplication of genotypes, and energy management. We also set the rules of organism building. It would be difficult to simulate a world with quarks, atoms or even proteins as its basic elements. There would be too many of them regarding reasonable size of the artificial world and computational requirements. That is why the basic element of our organisms is much bigger – it is a rod (bar, stick, cylinder...). Such an element can be assigned various functions depending on its genetic description: it can be just a stick, or can transmit and process signals and therefore be a part of a "brain", or be a receptor, or can have "muscles" and cause moves, or can be specialized in supplying energy.

A group of connected sticks makes up an independent organism. It becomes alive when put into the simulator!

The physical simulation module computes interaction of an organism with the world (*"Framsworld"*), analyzes forces influencing particular sticks and computes their new positions. The simulation takes place in a three-dimensional space, and uses finite elements theory and rigid bodies dynamics.

The neural module computes excitations in neural nets, collects data from receptors and sends signals to muscles. Organisms' neuron nets are different from those usually used in AI because of their free topology and inertia of neurons.

The energetic module analyzes gains and losses of energy. From settings in an experiment definition an organism can, for example, gain energy by assimilation or absorption of food, and use it for the work of muscles and neurons. After using up all its energy an organism may "die".

The creation module creates new organisms, for example by mutating and crossing over genotypes of the best creatures which have lived so far (ancestors).

The simulator is controlled by the script which defines the experiment. A sample experiment is "standard.expdef".

4 Simulation Details

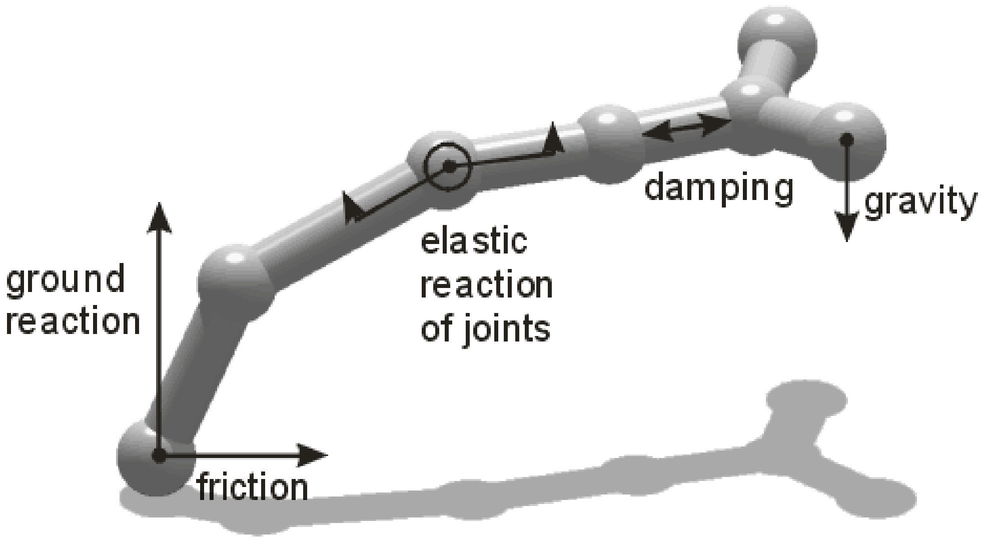
4.1 Physical (creature's body) simulation

Bodies of creatures are divided into small pieces (at sticks' ends) which are ideal material points. This approach is called "finite element method": not every point of a material body is simulated – only a finite number of points, representing small volumes in the body. The simulator calculates all the forces affecting a given point: gravity, elastic reaction when joined with other points, ground reaction and friction when touching the ground, etc.

In our model some assumptions are taken to simplify calculations. A primitive but fast numeric integration method is used, so the results are not very exact when dealing with big forces.

The picture on the right shows sample forces calculated in the Framsticks physical simulator, *Mechastick*.

The body is made from **parts** (points) and **joints** (sticks, rods).



Another simulation engine that can be used within Framsticks is ODE.

4.2 Neural (creature's brain) simulation

Neural network is made from neurons and connections. The Framsticks simulator supports many types of neurons (for example sigmoid neuron **N**, noise generator **Rnd**, differential neuron **D**, delay neuron **Delay**, threshold neuron **Thr**), and users can easily create their own signal processing neurons using FramScript and editing `scripts/*.neuro` files.

4.2.1 Formulas for neurons: N, Nu, D, Thr, *, Rnd, Sin, Fuzzy

Note: For most neurons, multiple inputs do not have idividual meaning and are aggregated with respect to their weights (weighted sum is computed).

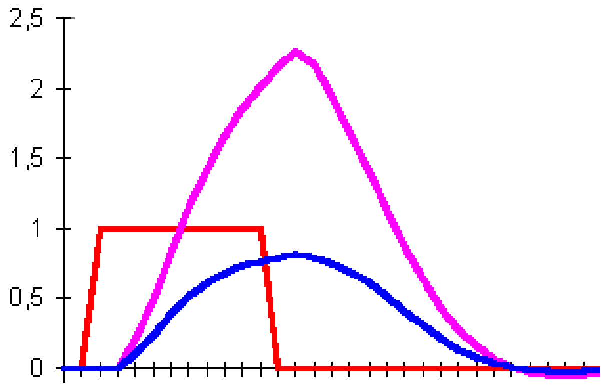
Neuron type	How simulated

N and **Nu** (sigmoid neurons)	In each simulation step: `#define NEURO_MAX 10.0` `input=getWeightedInputSum();` `velocity=force*(input-state)+inertia*velocity;` `state+=velocity;` `if (state>NEURO_MAX) state=NEURO_MAX;` `else if (state<-NEURO_MAX) state=-NEURO_MAX;` `tmp=state * sigmo;` **N**: `if (tmp<-30.0) output = -1;          //safer than exp(-tmp)` `else output = (2.0/(1.0+exp(-tmp))-1.0); // -1..1` **Nu**: `if (tmp<-30.0) output = 0;           //safer than exp(-tmp)` `else output = (1.0/(1.0+exp(-tmp)));     //  0..1` • `state`, `velocity` – internal variables • `sigmo`, `force`, `inertia` – neuron parameters (properties) • `input`, `tmp` – temporary variables
D (differentiate)	Calculate the difference between the current and the previous input values.
Thr (threshold)	Outputs `lo` if the input signal is below the `t` threshold. Outputs `hi` otherwise. `t`, `lo` and `hi` are parameters.
***** (constant output)	Outputs the value of 1.
Rnd (random noise)	Outputs random values (uniform distribution) in the range of –1..+1
Sin (sinus generator)	The output sinusoid has frequency `f` and initial phase t. The output frequency depends on the neuron input value. Parameters: • f0 (base frequency). The output frequency `f` = f0 + current_neuron_input_value. • t (time) – initial phase.
Fuzzy (fuzzy control system)	See this paper for details; see this movie for demonstration.

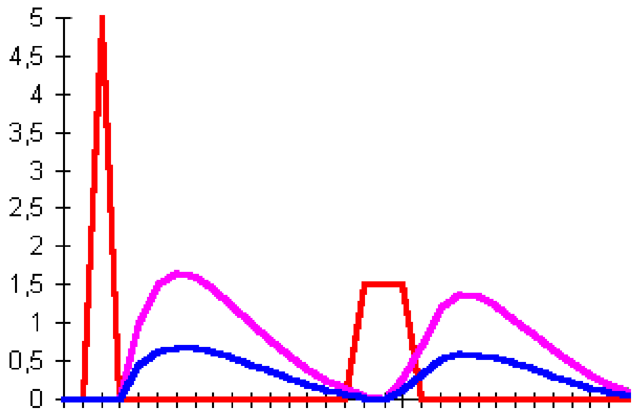
4.2.2 The N neuron in detail

Sigmoid neurons (with short name **N**) use a simple weighted sum of input signals. Excitation influences neuron state, which has some inertia. Stronger signals can change the state faster than weak signals. Output is flattened to a [-1,+1] range using basic sigmoidal function. See examples below: (input/state/output)

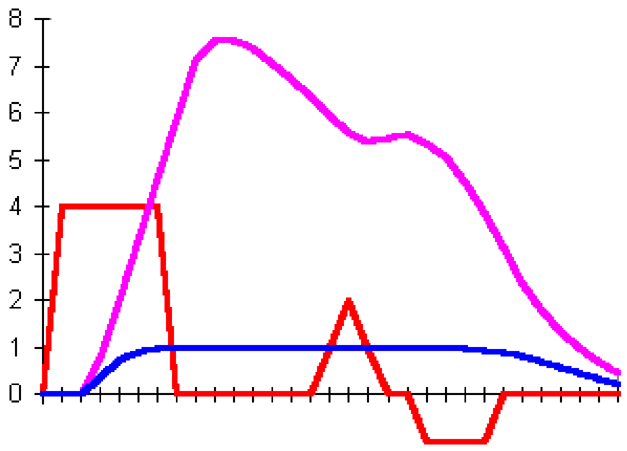
Simple excitation. The state goes up when the input is positive and falls down when theinput reaches zero. Note that in this example the neuron's state can fall below zero due to its inertia.



Short but strong impulse gives similar results to a weak and long one.



In this example a strong signal causes saturation of the neuron (its state goes very high). Later signal changes do not influence the output.



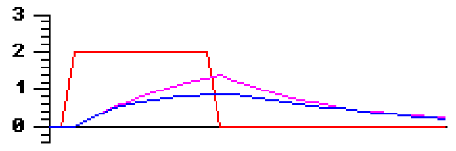
4.2.2.1 N parameters

The 'N' neuron has three properties (parameters) which influence its behavior:

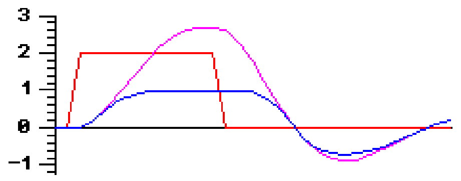
- *Force*, noted as 'fo', value range: 0..1, default 0.04
- *Inertia*, noted as 'in', value range: 0..1, default 0.8
- *Sigmoid*, noted as 'si', any real number accepted, default 2.0

Force and *inertia* influence changes of the inner neuron state. In each simulation step, the neuron state is modified towards the value calculated from input excitations. *Force* determines how fast the value is changed. Maximum value of 1.0 gives instant reaction. Low values, like the default (0.04) cause smooth 'charging' and 'discharging' of the neuron. Neuron's *inertia* is similar to the physical inertia of a body: it sustains its state change tendency. Low *inertia* values have very little influence on the state. Values near the maximum (1.0) can result in oscillations of the neuron state. The following pictures show sample usage of these parameters (input/state/output).

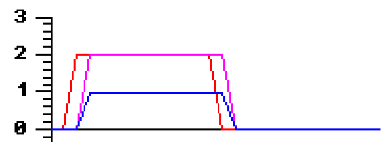
Force=0.1, *inertia*=0
Slow state change. Note the instant reaction after the input signal pulse when inertia is disabled.



Force=0.1, *inertia*=0.8
With inertia enabled, the neuron's state rises above the input pulse amplitude, and then drops below zero. The final state is achieved after several oscillations.



Force=1, *inertia*=0
Maximum force coefficient results in an instant input to output propagation.



The third, *sigmoid* coefficient changes the output function. Detailed formulas which describe the work of the **N** neuron are as follows:

$$velocity_t = velocity_{t-1} \cdot inertia + force \cdot (input_t - state_{t-1})$$

$$state_t = state_{t-1} + velocity_t$$

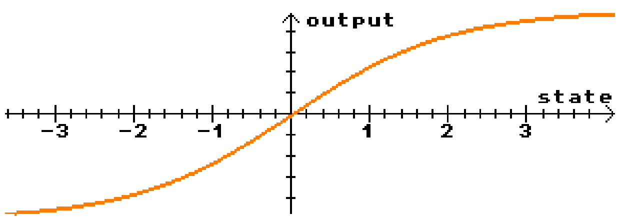
$$output_t = \frac{2}{1 + e^{(-velocity_t \cdot sigmoid)}} - 1$$

where

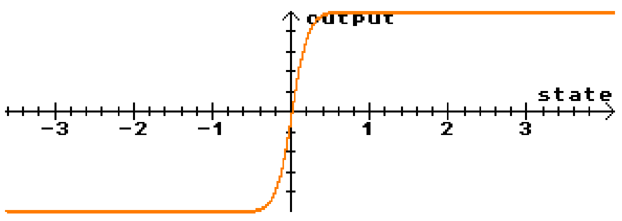
input: weighted sum of neuron inputs,
velocity: analogous to physical velocity,
state: internal state (analogous to physical location),
output: output signal.
Subscripts represent the time moment.

The following pictures show sample usage of the *sigmoid* parameter.

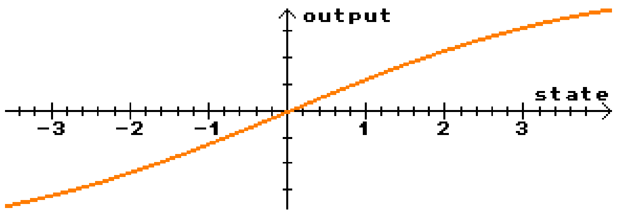
Sigmoid=2.0
Default.



Sigmoid=10.0
High values nearly produce a threshold function.



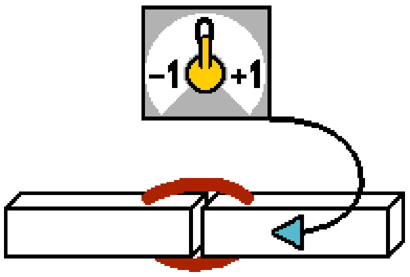
Sigmoid=0.5
Low values give a nearly linear output function.

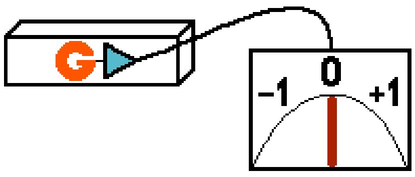


4.3 Special neurons: muscles and receptors

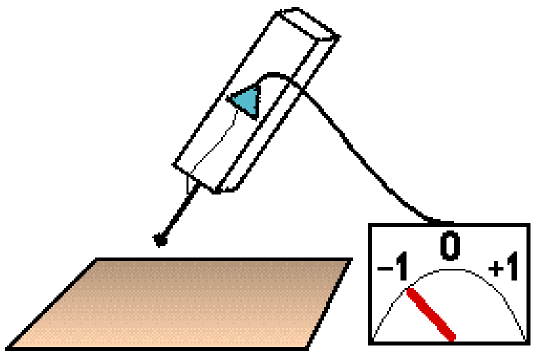
Basic muscles (actuators) and receptors (sensors) are illustrated below.

A muscle neuron ('|' or '@' in genotype) can change the relative orientation of the controlled stick (relative to the previous stick).
This simple 2-stick creature with a bending muscle in the middle is described by the genotype "XX[|...]". The joint stays straight when the signal is equal to 0. Positive and negative values bend the constructon in the opposite directions. [animation]

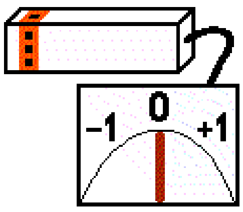




The "G" receptor (gyroscope, equilibrium sense) gives information about the stick's orientation relative to the gravity force. A gyroscope mounted on a horizontally aligned stick sends 0 to its outputs. Vertical position is perceived as -1 or +1 depending on which end is higher. [animation]



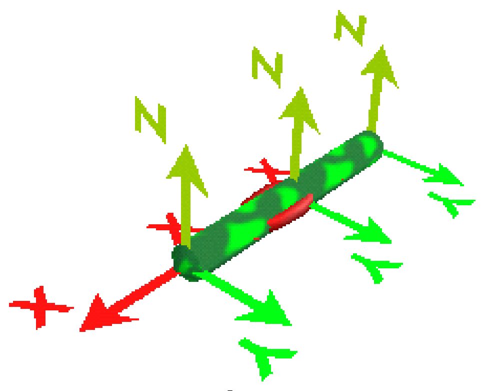
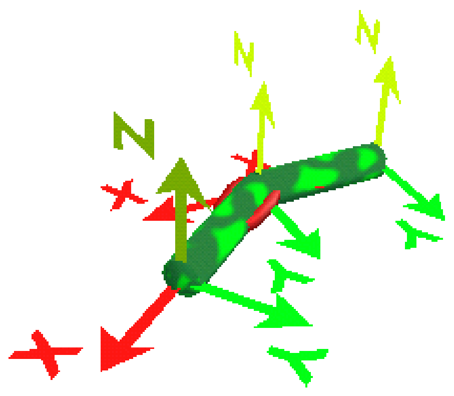
The "T" receptor (touch) can be imagined as a whisker attached to the stick. Its relaxed state is -1 (nothing detected in the whiskers' range). The signal value grows as the stick gets closer to any material object. Reaches 0 when the stick touches the ground. Higher values mean that the stick is pushed into the ground. [animation]



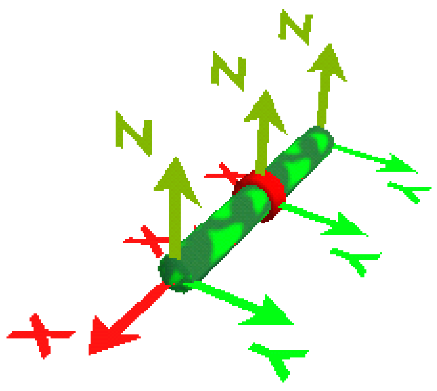
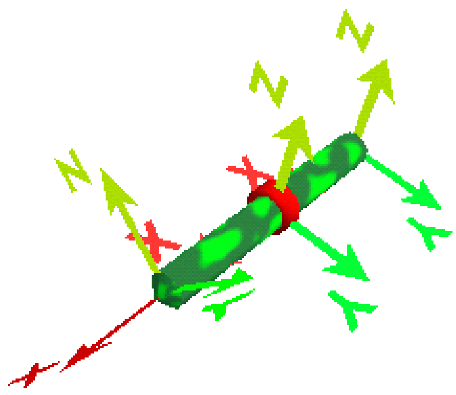
The "S" receptor (smell) excitation depends on the sum of the neighbor energy sources (energy balls and other creatures). Output range is from 0 (nothing detected) to 1 (maximum). Rich or closer energy sources smell 'stronger' than small or distant ones. [animation]

4.3.1 Formulas for basic effectors and receptors

Neuron type	How simulated

| (muscle)
 	
[animation]
	Changes the relative orientation of the joint's parts by rotating the second part around the first part's Z axis. Parameters:
	
	• p (power) – muscle strength. Affects the movement velocity and the maximum force the muscle can exert.
	• r (range) – Affects movement range, maximal value 1.0 means full range (-180°..+180°). The following rule applies:
	
	`Angle = Range * Signal * 180°`
	
	Sample creature (***f1*** genotype `XX[|]`)
	
	`//0`
	`p:`
	`p:1, m=2`
	`p:2`
	`j:0, 1, dx=1`
	`j:1, 2, dx=1`
	`n:j=1, d=|`
	
	
	Neutral position
	Singal = 0
	Angle = 0°
	
	
	Muscle active
	Signal = 0.2
	Angle = 36°
	
	Explanation: The joint is attached to parts #1 and #2. Part #2 was rotated around the Z axis of Part #1, so its position and orientation was changed.
	
	The above pictures were created using the "standard-xyz" OpenGL visualization style, which can be used to investigate part orientations in creatures.

@ (muscle)	Changes the relative orientation of the joint's parts by rotating the second part around the first part's X axis. Parameters: • p (power) – muscle strength. Affects the movement velocity and the maximum force the muscle can exert. This musle always uses the full movement range (-180°..+180°).

	`Angle = Signal * 180°`
	
	Sample creature (***f1*** genotype `XX[@]`)
	
	`//0`
	`p:`
	`p:1, m=2`
	`p:2`
	`j:0, 1, dx=1`
	`j:1, 2, dx=1`
	`n:j=1, d=@`
	
	
	Neutral position
	Singal = 0
	Angle = 0°
	
	
	Muscle active
	Signal = 0.25
	Angle = 45°
	
	Explanation: The joint is attached to parts #1 and #2. Part #2 was rotated around the X axis of Part #1. Part #2 was not moved, because it lies on the X axis (in creatures built from ***f1*** genotypes, all parts lie on the X axis).
	
	The above pictures were created using the "standard-xyz" OpenGL visualization style, which can be used to investigate part orientations in creatures.

G (gyroscope) [animation]	state = (part1.z – part2.z) / stick_length
T (touch) [animation]	• if touches: state = distance of part and ground (which is equivalent to positive depth) • if does not touch: check along x orientation of the part ♦ state = –1.0 if there are no objects closer than 1.0 distance ♦ state = ... (intermediate negative values) ♦ state = 0.0 if a touched object is just at the **T** part
S (smell) [animation]	state = (sum_all_energy_sources[energy / distance^2])/100 (if distance<1, then use 1)

Water (water detector / water pressure indicator) [animation]	• above surface: state = 0.0 • below surface: state = depth
Energy (energy level) [animation]	• state = 1.0 for initial energy level (a newborn organism) • state = ... (intermediate values) • state = 0.0 for no energy (creature dies)

4.4 All neuron types: summary

N (Neuron)
 	
Standard neuron
	`supports any number of inputs`
	`provides output value`
	`does not require location in body`
	
	`Properties:`
	`Inertia (in) float 0..1`
	`Force (fo) float 0..999`
	`Sigmoid (si) float -99999..99999`
	`State (s) float -1..1`

G (Gyroscope)
 	
Equilibrium sensor.
0=the stick is horizontal
+1/-1=the stick is vertical
	`does not use inputs`
	`provides output value`
	`should be located on a Joint`

T (Touch)
 	
Touch sensor.
-1=no contact
0=just touching
>0=pressing, value depends on the force applied
	`does not use inputs`
	`provides output value`
	`should be located on a Part`

S (Smell)
 	
Smell sensor. Aggregated "smell of energy" experienced from all energy objects (creatures and food pieces). Close objects have bigger influence than the distant ones: for each energy source, its partial feeling is proportional to its energy/(distance^2)
	`does not use inputs`
	`provides output value`
	`should be located on a Part`

* (Constant)	`does not use inputs`

Constant value	provides output value does not require location in body
| (Bend muscle)	uses single input does not provide output value should be located on a Joint Properties: power (p) float 0.01..1 rot.range (r) float 0..1
@ (Rotation muscle)	uses single input does not provide output value should be located on a Joint Properties: power (p) float 0.01..1
D (Differentiate) Calculate the difference between the current and previous input value. Multiple inputs are aggregated with respect to their weights	supports any number of inputs provides output value does not require location in body
Water (Water detector) Output signal: 0=on or above water surface 1=under water (deeper than 1) 0..1=in the transient area just below water surface	does not use inputs provides output value should be located on a Part
Energy (Energy level) The current energy level divided by the initial energy level. Usually falls from initial 1.0 down to 0.0 and then the creature dies. It can rise above 1.0 if enough food is ingested	does not use inputs provides output value does not require location in body
Ch (Channelize) Combines all input signals into a single multichannel output; Note: ChSel and ChMux are the only neurons which support multiple channels. Other neurons discard everything except the first channel.	supports any number of inputs provides output value does not require location in body
ChMux (Channel multiplexer)	uses 2 inputs provides output value

Outputs the selected channel from the second (multichannel) input. The first input is used as the selector value (-1=select first channel, .., 1=last channel)	`does not require location in body`

ChSel (Channel selector)
 	
Outputs a single channel (selected by the "ch" parameter) from multichannel input
	`uses single input`
	`provides output value`
	`does not require location in body`
	
	`Properties:`
	`channel (ch) integer`

Rnd (Random noise)
 	
Generates random noise (subsequent random values in the range of -1..+1)
	`does not use inputs`
	`provides output value`
	`does not require location in body`

Sin (Sinus generator)
 	
Output frequency = f0+input
	`uses single input`
	`provides output value`
	`does not require location in body`
	
	`Properties:`
	`base frequency (f0) float -1..1`
	`time (t) float 0..6.28319`

Delay (Delay)	`uses single input`
	`provides output value`
	`does not require location in body`
	
	`Properties:`
	`delay time (t) integer 1..1000`

Nn (Noisy neuron)
 	
Propagates weighted inputs onto the output, but occassionally generates a random value
	`supports any number of inputs`
	`provides output value`
	`does not require location in body`
	
	`Properties:`
	`Error rate (e) float 0..0.1`

Sf (Smell food)
 	
Detects only food, not other creatures
(in experiments with food in group #1)
	`does not use inputs`
	`provides output value`
	`should be located on a Part`

Thr (Threshold)
 	
if (input>=t) then output=hi else output=lo
	`uses single input`
	`provides output value`
	`does not require location in body`
	
	`Properties:`
	`threshold (t) float`
	`low output value (lo) float`
	`high output value (hi) float`

4.4.1 Experimental neurons

Do not use unless you know what you are doing. Some of these neurons can be unstable, many are used for special purposes.

Nu (Unipolar neuron [EXPERIMENTAL!]) Works like standard neuron (N) but the output value is scaled to 0...+1 instead of -1...+1. Having 0 as one of the saturation states should help in "gate circuits", where input signal is passed through or blocked depending on the other singal.	supports any number of inputs provides output value does not require location in body Properties: Inertia (in) float 0..1 Force (fo) float 0..999 Sigmoid (si) float -99999..99999 State (s) float -1..1
Fuzzy (Fuzzy system [EXPERIMENTAL!]) Refer to publications to learn about this neuron	supports any number of inputs provides output value does not require location in body Properties: number of fuzzy sets (ns) integer number of rules (nr) integer fuzzy sets (fs) string fuzzy rules (fr) string
VEye (Vector Eye [EXPERIMENTAL!])	uses single input provides output value should be located on a Part Properties: target.x (tx) float target.y (ty) float target.z (tz) float target shape (ts) string perspective (p) float 0.1..10 scale (s) float 0.1..100 show hidden lines (h) integer 0..1 output lines count (each line needs four channels) (o) integer 0..99 debug (d) integer 0..1
VMotor (Visual-Motor Cortex [EXPERIMENTAL!]) Must be connected to the VEye and properly set up.	supports any number of inputs provides output value does not require location in body Properties: number of basic features (noIF) integer number of degrees of freedom (noDim) integer parameters (params) string
Sti (Sticky [EXPERIMENTAL!])	uses single input does not provide output value should be located on a Part

LMu (Length muscle [EXPERIMENTAL!])	`uses single input`
	`does not provide output value`
	`should be located on a Joint`
	
	`Properties:`
	`power (p) float 0.01..1`

5 Genotype encodings

The Framsticks simulator supports various genotype formats. A "format" is a language you can use to describe creatures. The properties of this language are extremely important from the viewpoint of evolution. If you are a Framsticks beginner, read about the ***f1*** format first. For some people, the ***f0*** format is even easier.

f0	A low-level format which allows for building any creatures (least restrictive)
f1	An easy-to-use recursive language
f4	Describes the way of growing an organism (developmental encoding). Learn about ***f1*** before reading ***f4***
f2	Describes how basic parts of an organism are joined
f3	Encodes ***f2*** in "biological" genes with codons etc.
f5	A variant of ***f4***
f6	Describes chemical/metabolic rules of growing
f7	Accepts any string of symbols, a "messy" genome
f8	A parametric Lindenmayer system (L-system) **-** rules of body and brain development
f0Fuzzy	Used for the evolution of fuzzy control systems embedded in f0 genotypes (see this paper and this movie)
...	other encodings are easy to add

The "Comparison of Different Genotype Encodings..." scientific paper contains more formal and detailed description of genetic formats, their properties, and mutation and crossover operators.

If you enter a genotype, you have to indicate its format. If it is, for example, ***f4***, you should start the genotype with **/*4*/** prefix. For multiline genotypes (like ***f0***), you can also use another style for such a prefix: **//0**. If there is no format prefix, ***f1*** is assumed.

Although some genotype formats may look complicated, they are easy to learn. Just open the "new genotype" window and experiment with genotypes. You can enter anything valid and get an immediate preview! Select genes to see which creature parts are created by these genes. Click on creature parts to see which genes created them.

6 Genotype (format *f0*)

"Format Zero" genotype is the construction plan for any Framsticks creature. More detailed information on how to use it for development of one's own stick objects is available in Framsticks SDK (Software Developer Kit). See also the FRED program. Fundamental information is provided below.

6.1 f0 Syntax

Each line in **f0** describes one object in the model. Lines starting with the '#' sign are considered to be comments and are ignored. The syntax is:

`CLASSID:PROPERTY1,PROPERTY2`,...

`CLASSID` - alphanumeric identifier of the class. Currently, there are four object classes (see f0 Semantics)

`PROPERTY1,PROPERTY2`,... - a set of properties of the object. Each class defines the **sequence** of properties and a reasonable **default value** for each one. The full definition of the property is `NAME=VALUE`. You can skip `NAME=` if the given property is next in the sequence. Object descriptions with omitted property names are not recommended as they may be misinterpreted by future GDK versions with different property sets. There is an exception for easier editing: skipping inside the "natural" (= "unlikely to change") property sequences like x,y,z is allowed, you can write "x=1,2,3" instead of "x=1,y=2,z=3". (Such exceptional properties are marked by flag 1024 in the property definition, see Param::flags(int i)). You can also skip the whole definition of a property and accept the default value. If you want to pass special characters (for ex. a comma) or set the empty string value (""), `VALUE` can be placed between quotes (").

Examples

Let's assume the following class definition: CLASSID=ob and four properties with default values: a=0, b=1, c=2, d=3

Some valid **f0** descriptions:

all properties are set explicitly	`ob:a=9,b=8,c=7,d=6`
you can omit property names if the sequence is obeyed	`ob:9,8,7,6`
some properties have no names, but they are deduced from the sequence	`ob:d=6,a=9,8,7`
"b" and "c" are not defined – default values will be used (b=1,c=2)	`ob:9,,,7`
b="", c=2	`ob:9,"",,7`
b="," (comma), c=2	`ob:9,",",,7`
default values used for "a","c" and "d"	`ob:b=8`
default values used for all properties	`ob:`

6.2 f0 Semantics

Parts, Joint and Neurons have **reference numbers** used to attach other objects. References start with 0 and every new object in the class gets the next reference number.

6.2.1 Part object

Creates instance of the Part object.

- **CLASSID=p**:
- **x=0,y=0,z=0** - position in 3D
- **rx=0,ry=0,rz=0** - 3D orientation (amount of rotation around each of 3 axes)
- **m=1** - physical mass
- **s=1** - size (for collisions)
- **dn=1** - density
- **fr=0.4** - friction
- **ing=0.25** - ingestion
- **as=0.25** - assimilation
- **i**="" - general purpose "info" field

The Part's position and orientation can be overriden by a joint definition using the "delta option". In this case, you can skip x, y and z in the part definition (see f0 Examples).

6.2.2 Joint object

Creates instance of the Joint object.

- **CLASSID=j**:
- **p1=-1,p2=-1** - ref.numbers of the two connected parts. Note that the default -1 is illegal here, you must not omit these properties. Both parts have to be created earlier.
- **rx=0,rx=0,rz=0** - rotation: can be used to enforce specific orientation of the second part (p2) relative to the first part (p1). Orientation can influence some of the effectors/receptors and can be used as growing direction while creature is being constructed.
- **dx=?,dy=0,dz=0** - delta option: if you specify any value for dx, all three deltas are used as displacement applied to the second part (p2) relative to the first part (p1). Local coordinate system of p1 is first rotated (as defined by [rx,ry,rz]) and then translated by [dx,dy,dz]. This technique allows you to define relative placement of parts. Without this delta option, absolute part positioning is used (as defined by the coordinates [x,y,z] of the part).
- **stif=1** - stiffness
- **rotstif=1** - rotation stiffness
- **stam=0.25** - stamina
- **i**="" - general purpose "info" field

6.2.3 Neuron object

Creates instance of the Neuro object.

A signal processing unit, sensor, or effector. Neuron reference number is used in f0docneuroitem definition.

- **CLASSID=n**:
- **p=-1** - ref.number of the part the neuron is attached to (the part has to be created earlier).
- **j=-1** - ref.number of the joint the neuron is attached to (the joint has to be created earlier).
- **d** - neuron class description, like "classname:properties_and_values"
- **i**="" - general purpose "info" field

6.2.4 Neuron input definition (connection)

Add input to the neuron = weighted connection to the other neuron. Both objects must be already created.

- **CLASSID=c**:
- **n** - ref.number of the parent neuron
- **i** - ref.number of the neuron to be connected as the parent's input
- **w=1.0** - connection weight (optional)

6.3 Model validity constraints

- at most one joint can directly link two parts
- each joint must be connected with two distinct parts
- all parts must be directly or indirectly connected with each other
- delta joints must not form cycles
- for each joint, the part-to-part distance must not exceed the value defined as the maximum Joint.dx (2.0)

6.4 f0 Examples

To see these in Framsticks application, you have to start writing **f0** genotype with "//0" (two slashes and zero and new line). The shortest **f0** genotype is a single Part:

```
p:
```

A single stick, "X" in **f1**

```
p:
p:1
j:0,1
```

Three sticks in line, "XXX" in **f1**, no "delta option" - absolute coordinates used in all parts

```
p:
p:1,m=2
p:2,m=2
p:3,
j:0,1
j:1,2
j:2,3
```

Three sticks line, "XXX" in **f1**, with "delta option" - relative positioning (dx=1)

```
p:
```

```
p:m=2
p:m=2
p:
j:0,1,dx=1
j:1,2,dx=1
j:2,3,dx=1
```

Three sticks star (120 degrees), "X(X,X)" in **f1**, no delta option, absolute coordinates are awkward and the sticks' length cannot be seen

```
p:
p:1, m=3
p:1.50017, -0.865927
p:1.50017, 0.865927
j:0, 1
j:1, 2
j:1, 3
```

Three sticks star (120 degrees), "X(X,X)" in **f1**, using delta option, dx=1, note that 120 degrees / 2 = 1.047 rad

```
p:
p:m=3
p:
p:
j:0, 1, dx=1
j:1, 2, rz=-1.047, dx=1
j:1, 3, rz=1.047, dx=1
```

Neuron net example, "X[|G:1,1:2.3][@-1:3.4,0:4.5,T:5.6]" in **f1**

```
p:
p:1
j:0, 1, dx=1
n:p=1
n:j=0, d="|:p=0.25,r=1"
n:j=0, d=G
n:p=1
n:j=0, d=@:p=0.25
n:p=1, d=T
c:0, 2
c:0, 3, 2.3
c:1, 0
c:3, 0, 3.4
c:3, 3, 4.5
c:3, 5, 5.6
c:4, 3
```

Cyclic structure, parts are connected 0->1->2->3->0. Not possible in **f1** or **f4** formats.

```
p:0,0
p:1,0
p:1,1
p:0,1
j:0,1
j:1,2
j:2,3
j:3,0
```

7 Genotype (format *f1*)

7.1 Creature body

Two basic symbols are:

- X – stick,
- () – branch.

The body structure is built like a tree: new sticks are joined with ends of the previous ones.

- X(X,X) means two sticks from one-stick root,
- X(X,X,X) means three sticks from one-stick root,
- X(X,X,) is also possible, as well as
- X(,X,,,X,,X,,)

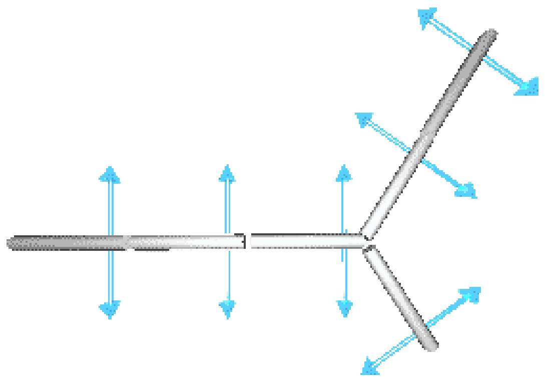
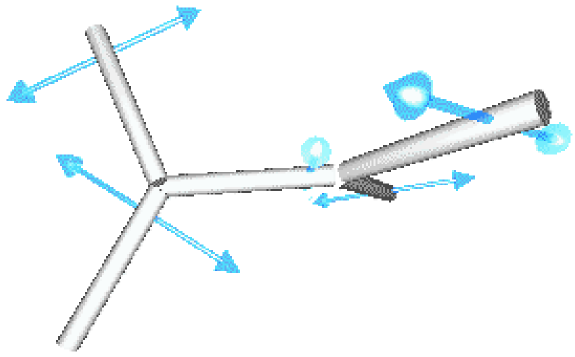
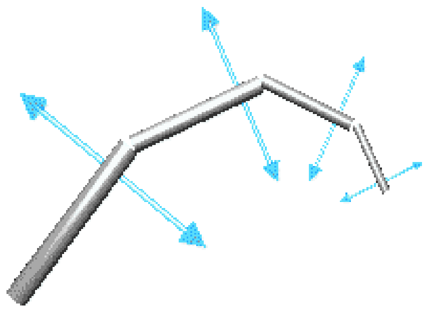
Inside parenthesis, the full angle is divided into as many parts as there are commas+2, and each stick in such a 'junction' has as much freedom.

Modifiers (special characters) can be placed before X's and ('s. They affect the following X and, usually less and less, further following X's. Modifiers modify stick position and its features. Big and small letters can be used; big letters increase the given property while small ones decrease it.

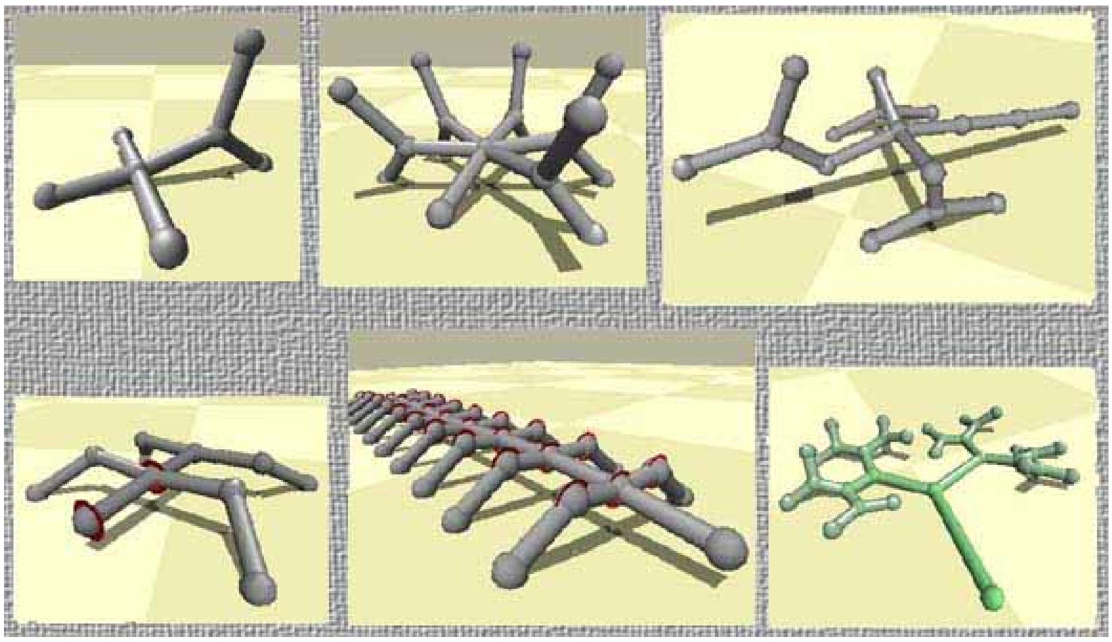
Modifiers: Rr, Qq, Cc, Ll, Ww, Ff, Aa, Ss, Mm, Ii, Ee.

Sticks' joints properties	
R	rotation (by 45 degrees) – this modifier DOES NOT affect further sticks
Q	twist
C	curvedness
Physical properties	
L	length
W	weight (in water environment light sticks swim on the surface)
F	friction (sticks will slide on the ground or stick to it)
Biological properties (mutually exclusive)	
A	assimilation = photosynthesis (a vertical stick can assimilate twice as much as a horizontal one)
S	stamina (increases chance of survival during fights, see also simulator parameters – destructive collisions)
M	muscle strength, a.k.a. muscle speed (strong muscles act with bigger force, gain higher speed, can resist bigger stress, and use more energy)
I	ingestion (ability to gain energy from food: energy balls or dead corpses)
Other	
E	energy (experimental).Creature's starting energy can be higher or lower when 'E' or 'e' is used. You should use it only when the "energetic efficiency mode" is enabled (see simulator parameters). Otherwise, disable 'eE' modifiers in genetic parameters!

7.1.1 Examples

XXX(XX,X)	
X(X,RRX(X,X))	
XlCXlCXlCX	

Some more samples of possible structures:



7.2 Creature brain

Neurons are placed in [], after X's. The following information can be supplied inside square brackets:

- neuron type
- properties (parameters) of the neuron
- inputs of the neuron (if it can have inputs)

The syntax is:

[*NeuronType***,** *PropertyAndInputList* **]** where *PropertyAndInputList* is a comma-delimited list of pairs *PropertyName***:***Value* and *NeuronInput***:***Weight*. If *NeuronType* is omitted, 'N' (standard neuron) is assumed.

Neuron inputs can be taken from different signal sources: other neurons' outputs or receptors. A neuron input can also have a constant value. If a neuron is a muscle, it can control its stick's bend or rotation.

The alternative (old, deprecated) syntax lets you describe two neurons (a muscle and 'N') in one:

[*MuscleType* *PropertyAndInputList* **]** where *MuscleType* is either '@' (stick's rotation) or '|' (stick bend). This old syntax creates a 'N' neuron, as described in [...], and a muscle of *MuscleType* with input from that neuron. Thus the old syntax is equivalent to
[N**,** *PropertyAndInputList* **] [** *MuscleType***,** -1:1 **]**

In the old genotypes, you could find names of 'N' properties noted as symbols !=/. They are deprecated, and equivalent to:

- '!' [exclamaton mark] – force (present property name: 'fo')
- '=' [equals] – inertia (present property name: 'in')
- '/' [slash] – sigmoid (present property name: 'si')

(see also the Simulation details section)

7.2.1 Examples

Enter these genotypes into Framsticks to see the corresponding neural networks.

- X[N]
- X[@]
- X[@][N]
- X[@,1:1.0][N]
- X[1:1.0][-1:2.0]
- X[1:1.0][G:2.0]
- X[N,fo:1,si:-4]
- X[Sin,f0:0.1,t:0.5]

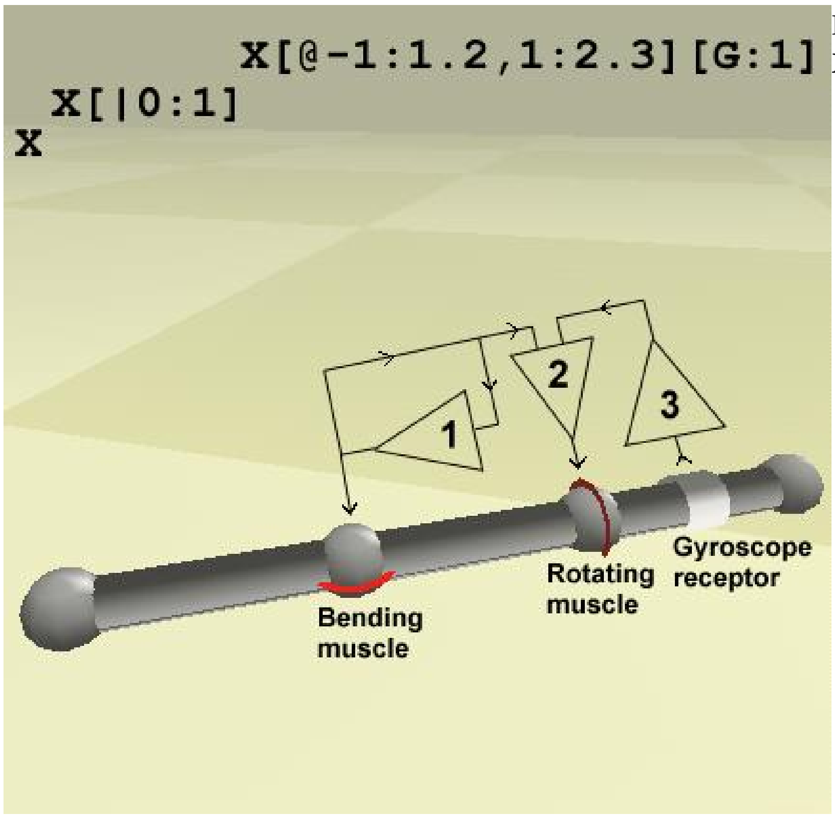
7.2.2 Examples of the old, deprecated syntax

... X[@-1:2,1:3] ...
means that a stick has one neuron 'in' it. It controls the stick's rotation (@), and the neuron has two inputs:

one comes from -1 relative position in the genotype, the other from +1 rel. position. The first 'signal' weight is 2, the other is 3.

X[|*:1,G:2]
is a stick with one neuron controlling its bend, having two inputs: one constantly equal to 1 and the other connected to a gyroscope (placed on the stick) weighted 2.

One stick can have many neurons – X[.....][.....][.....]



Example:
X X[|0:1] X[@-1:1.2,1:2.3][G:1]

- neuron (1) affects stick's bend and is looped recursively
- neuron (2) affects stick's rotation, and receives signals from neurons (1) and (3)
- neuron (3) has one input: gyroscope (equilibrium sense of the last stick)

8 Genotype (format *f4*)

This encoding scheme is an *indirect developmental encoding*, devised by Adam Rotaru-Varga.

Contents:

8.1 Overview

The ***f4*** encoding resembles the ***f1*** encoding, with an important conceptual difference: ***f1*** is composed of codes which are interpreted as structural elements (sticks, neurons), or their attributes. On the other hand, ***f4*** codes are interpreted as *instructions* to cells. An ***f4*** genotype describes the developmental process of an organism. Development starts with a single ancestor cell, which starts to execute instructions from the start of the genetic code. As the cell divides, new cells are created, which execute different instructions in parallel (differentiation). The development stops when all cells mature, and the final shape of the creature is the result of the whole development process.
Developmental encoding models biological growth of an individual. Some features of a developed creature are not encoded by a particular gene, but by an interplay between developing parts. These interactions are modeled during the development process.

The developmental process. A developing creature is composed of interconnected **cells**. A cell is either a stick, or a neuron, or still undifferentiated -- this is the **type** of the cell. Undifferentiated cells can turn into a stick or neuron (as a result of developmental instructions), but not the other way around.
Development starts with a single, undifferentiated ancestor cell, executing the start of the genotype.
During each step, each cell executes one (or more) instructions, in parallel. A new step is started whenever a division or a change in type occurs. Development stops when all cells stop changing. At this point, there should be no undifferentiated cells.

The minimal example. The minimal ***f4*** genotype looks like this: '**/*4*/ X**' (ex0). This says to the ancestor cell to turn into a stick (X), and stop development. The end result is a single stick -- corresponding to ***f1*** genotype 'X'.

Two-sticks example. /*4*/ <X>X (ex1). This looks more interesting: '&lt'; denotes cell division. It will create two cells, the first will execute the instruction immediately following the '<', while the other will execute the instruction after the corresponding '>'. Now, it looks like '<' and '>' act like parantheses, but that's only superficial. In fact, '>' means 'stop development'.
The exact process looks like this:.
step 0. Initially, there is the ancestor cell, no 1, undifferentaited.
step 1. cell 1 executes '<' (division), creates cell 2 undifferentaited.
step 2. cell 1 executes 'X', turns into a stick.
(cont) cell 2 executes 'X' (the other one), turns into a stick.
step 3. cell 1 executes '>' -- stops development.
(cont) cell 2 stops development.

The end result is two connected sticks, the same structure created by ***f1*** genotype 'XX'.

8.2 Details

An ***f4*** genotype is identified by the prefix '**/*4*/** '. This is not part of the genotype, but a comment identifying that it is of type ***f4***.

Most ***f4*** codes are one-letter codes, with some exceptions. An ***f4*** genotype is expressed as a string of codes. However, because division branches the sequence of instruction in two, the genotype can be conceptualised as a binary tree of codes. The string representation corresponds to a pre-order traversal. (Note that an ***f1*** can be conceptualised as a tree as well.)

Codes are the following:

'<' Division. Creates a new cell. The new cell will be connected to the old one. This is the only way of creating a new cell.
After division, the two cells will execute different codes. The '<' is followed by the codes executed by the first cell (ending in a '>'), and then the codes to be executed by the second cell. Thus the code to be executed by the second cell can be found after the corresponding '>'. Note that both code sequences can contain further divisions. The general form is:
< ...cell 1 code... > ...cell 2 code... >
If there are **n** divisions in a genotype, they will create **n+1** cells. This also means **n+1** cell-stop markers '>'. However, the *very last '>' can be omitted*. This is because the last stop is not followed by anything else, and to have an equal number of '<' and '>' codes.
Example: <LL<X>X>llX (ex53)
In this case, after the first division, the first cell will continue with the 'L' (pos 2.), while the second, freshly created cell with the 'l' (pos 9).
Usually undifferentiated cells divide, and later differentiate into sticks or neurons. However, a neuron can divide, and in this case the new cell will be a neuron as well, with the same characteristics as the old cell. Existing links are also duplicated. Sticks cannot divide.

'X' Turn into stick. Turns the cell into a stick. The cell must be undifferentiated. It will remain a stick, since a stick cannot change its type (nor divide).

'N' Turn into neuron/muscle. Turns the cell into a neuron. The cell must be undifferentiated. It will remain a neuron, since a neuron cannot change its type.

'>' Stop development of cell. Cell should not be undifferentiated. This symbol also has the role of a delimiter.

',' Increase branching angle (comma). Increase the branching count (angle) of future divisions. Changes takes effect when the dividied daughter cell turns into a stick. Cell can be undifferentiated or a stick. Example: ex3.

'L'/'l' Increase/decrease length of stick. L: increases length by 0.3*(2.5-len). l: decreases length by 0.3*(len-0.3). Works with sticks and undifferentiated cells.

'R'/'r' Increase/decrease rotation by 45 degrees. Works with sticks and undifferentiated cells.

'C'/'c' Increase/decrease curvedness.

'Q'/'q' Increase/decrease twist.

Neural parameters can be modified by a sequence of increasing/decreasing codes. (Note that in f1 these are set using concrete numerical values.) These codes are of the form ':X+:' or ':X-:' for increasing and decreasing respectively **':!+:' Increase neural force**, by (1.0 – force) * 0.2.

':!-:' Decrease neural force, by force * 0.2.

':=+:' Increase neural inertia, by (1.0 – inertia) * 0.2.

':=-:' Decrease neural inertia, by inertia * 0.2.

':/+:' Increase neural sigmoid value, multiply by 1.4.

':/-:' Decrease neural sigmoid value, divide by 1.4.

'[...:...]' Add a neural link. Adds a neural link to a neuron (cell must be neuron). The format is: '[' (input link) ':' (weight) ']'.
Input link is an integer number, interpreted as the relative reference to the neuron where the link is originating. 1 means the 'next' neuron, -1 the previous, 0 this one, 3 three from here on, and so on. Note that relative references are based on the *current* structure, and not the final one!
There are special input codes for senses: 'G' for gyroscope (adds a gyroscope sense to the stick the neuron is connected to), 'T' for touch sensor, and 'S' for smell sensor.
The weight is a real number representing the weight of the link. Examples: ex9, ex11.

'|' Turn into a bending muscle. Turns the cell into a bending muscle. The cell must be a neuron. Examples: ex10, ex51.

'@' Turn into a rotating muscle. Turns the cell into a rotating muscle. The cell must be a neuron. Examples: ex11, ex52.

Codes that affect biological properties:

'A'/'a' Increase/decrease assimilation.
'I'/'i' Increase/decrease ingestion.
'S'/'s' Increase/decrease stamina.
'M'/'m' Increase/decrease muscle strength.
'F'/'f' Increase/decrease stick strength.
'W'/'w' Increase/decrease stick mass.
'E'/'e' Increase/decrease stick energy.

'#' Repetition marker. This code allows certain other codes to be repeated more than once. The explanation of this code requires certain general details, left out from the discussion so far for the sake of clarity. This code is quite tricky, but it is also powerful: it can create repetitions of the same codes (and thus substructures) withouth the dupication of the codes themselves.
Each cell has a pointer to the currenty executing code. But they have another pointer, which is the 'to repeat' pointer. They also have an associated 'to repeat' counter. The '#' code creates a branching in the genotype tree, much like the '' as well: if the repeat counter is 0, it means regular 'stop', as described above. However, if the counter is not 0, it is decremented, and if it still not zero, the cell 'jumps' back to the repeat pointer. If the repeat counter gets to 0, the second child of the '#' is executed to finish off.
What all this means is that a genotype of the form
#n ...repcode... > ...endcode...
means that the repcode part will be repeated n times, and the endcode once in the end.
There's one more detail: when a cell divides ('new cell inherits the repeat counters, and not the old one. Thus

only one cell will continue the repetition.
Examples: ex8a, ex15. ex16. ex17.

8.3 Genetic operators

Mutation produces some localised changes in an ***f4*** genotype. Mutation operates on the tree representation of a genotype. A single mutation can change a code, change a parameter of a code, add a new code (division, neural link, repetition marker, or simple code), or delete a code. A mutation operator can consist of several single mutations. The amount of change of a mutation is estimated from the ratio of number of changed codes and total number of codes.

Crossover operates on ***f4*** genotype trees, and exchanges two subtrees of two genotypes. The size of the subtrees varies, between 10 and 90 percent of the genotype.

8.4 Examples

example no.	***f4*** genotype	corresp. ***f1*** genotype (approx)	description
ex0	X	X	a single stick
ex1	<X>X	XX	two sticks connected one-after-the-other
ex2	<<X>X>X	X(X,X)	two sticks connected to the same third one (branching)
ex3	<,<,,<,X,,>X>X>X	X(,X,,,X,,X,,)	a 3-way branching with different brancing angles
ex4	<X><X><<X>X><X>X	XXX(XX,X)	more branching
ex5	<<X><<X>X>X>X	X(X,X(X,X))	more complex branching
ex6	<X>l<X>l<<X>X>LLLX	XlXlX(LLLX,X)	different length factors
ex7	<<X>RR<<X>X>X>X	X(X,RRX(X,X))	branching with rotation R
ex8a	#3<X>lC>X	XlCXlCXlCX	repetition, C
ex8b	<X>lC<X>lC<X>lCX"	XlCXlCXlCX	curvedness C
ex9	<X><N[1:2]>N[-1:3]	X[1:2][-1:3]	a stick with two neurons attached to it. The two neurons are connected to each other
ex10	<X><X><N|[1:2]>N[-1:3.5]	XX[|1:2][-1:3.5]	a structure with two neurons and a bending muscle
ex11	<X><<,<X,><N@[1:20]> N[G:30]>X>X	XX[@1:20][G:30](X,,X,)	a structure with a rotating muscle and a gyroscope sense
ex12	<X>N|[*:1][G:2]	X[|*:1,G:2]	

			a stick with a bending muscle
ex13	<X>N<[*:0]>[-1:10]<>	X[*:0][-1:10][-2:10]	duplicating links upon neuron division
ex14	<X><<X>N|[0:1]><X><N@[-1:1.2][1:2.3]>N[G:1]	XX[|0:1]X[@-1:1.2,1:2.3][G:1]	various neurons and links
ex15	#10,<<X>X>>LLX	X(,,X(,,X(,,X(,,X(,, X(,,X(,,X(,,X(,,X(,LLX, X)X)X)X)X)X)X)X)X)X)	repeating division 10 times
ex16	#10,<<X><<X>X>X>>LLX	X(,X(,X(,X(,X(,X(,X(, X(,X(,X(,LLX,X(X,X)), X(X,X)),X(X,X)),X(X,X)), X(X,X)),X(X,X)),X(X,X)), X(X,X)),X(X,X)),X(X,X))	even more repetitions
ex17	rr<X>#9<,<X>RR<<llX>LX>LX>>X	rrXX(,X(,X(,X(,X (,X(,X(,X(,X(,X, llRRX(LLX,LLX)), llRRX(LLX,LLX)), llRRX(LLX,LLX)), llRRX(LLX,LLX)), llRRX(LLX,LLX)), llRRX(LLX,LLX)), llRRX(LLX,LLX)), llRRX(LLX,LLX)), llRRX(LLX,LLX))	a worm-like creature, composed of repeated identical segments ('kukac')
ex51	<X><X>N|	XX[|] /* ?? */	two sticks and a bending muscle between them.
ex52	<X><X>N@	XX[@] /* ?? */	two sticks and a rotating muscle between them.
ex53	<LL<X>X>llX	LLX(llllLX,X)	3 sticks in a Y formation

Related links

- ***f1*** genotype format
- All Framsticks genotype languages

9 Simulation Parameters

If you need help, read the hints (tooltips) which appear when you stop the mouse pointer over a name of a parameter in the program. Documentation below is provided in addition to those hints and discusses only some issues.

See also:

Tips for users
Interface
Interface parameters

This page:

9.1 Experiment

Experiment definition: choose experiment definition which controls behavior of the Framsticks system. It is a script file with the ".expdef" extension. All available experiment definitions are summarized here. Advanced users can create their own experiment definitions and thus exploit the full potential of the system. Press "Apply" each time you change the experiment definition.

Parameters of the **standard** experiment definition are available here (see also tips).

Two experiment definitions more:

- **standard-log.expdef** in scripts_sample subdirectory (provided as an example of logging all genetic operations).
- **standard-tricks.expdef** in scripts_sample subdirectory (provided as an example of complex fitness functions with customized performance measurements, and changing creature locations during their simulation).

Initialize experiment: perform initialization actions for the selected experiment definition (for example, create appropriate gene pools and populations, clear them, insert the initial genotype, reset counters, etc.). It is recommended to perform initialization before running an experiment.

Experiment: Gene pools: <gene pool name>

An experiment definition may group genotypes into "gene pools". Each of them has separate settings. Settings affect all genotypes belonging to a given group (gene pool).

Fitness formula: advanced users can enter a custom fitness formula here. For the "standard" experiment definition, the formula is automatically created when you change weights of fitness criteria, and it reflects those settings.

Scale fitness?: if turned on, fitness is modified linearly according to the scaling rules. Thus final fitness is computed, which is equal to:

- 1.0 for creatures with average fitness
- 0.0 for creatures with fitness less or equal to the value of (average fitness – Shift coefficient * standard deviation of fitness)
- Scaling coefficient for creatures with maximal fitness.

Experiment: Populations: <population name>

An experiment definition may group simulated objects into "populations". Each of them has separate settings. Settings affect all individuals/objects which are simulated and belong to a given group (population).

Energy calculation: calculate energy balance during simulation?

Death: remove objects when their energy reaches 0 (creatures die)?

Neural net simulation: after a creature is put into the world, activate its neural network

- Off – don't simulate neural networks at all
- Immediately
- After stabilization – start brain simulation when the creature completely stops (all initial vibrations stop).

Performance sampling period: especially important for estimating distance and velocity. Low values cause vibrations/little movements to be counted as velocity. High values let only smooth, straight moves to be counted.

Performance calculation: after a creature is put into the world, calculate its performance

- Off – don't calculate at all
- Immediately
- After stabilization – count measurements when the creature completely stops (all initial vibrations stop).

NOTES. If you want to evolve quick creatures, calculating their speed including the result of falling down and turning over would be unfair. This is why "delay for stabilization" may be introduced. Otherwise, creatures could gain additional ("unfair") fitness bonus because of their initial position (creatures fall onto the ground and earn speed without using neurons).

Speed and movements are measured using center of gravity position.

Consider the following special (educational) situations. (1) When a creature waits for stabilization, other living and active creatures may push it and move it, so that it cannot stabilize. (2) When you set immediate simulation of neural network and after-stabilization performance calculation, a creature might start moving immediately and would never stabilize. Thus performance and energy wouldn't be calculated, and such a creature would live forever as long as it kept moving. (3) In water environment, stabilization may be difficult to achieve.

Muscle static work, Muscle dynamic work, Assimilation productivity: these settings influence energy balance. When non-zero values are used, creatures may spend more energy than Idle metabolism, and can gain energy by assimilation.

9.2 Files

Save backup: if you set this value to a number N>0, the simulator writes an EXPT (experiment state) file every N events. An "event" is triggered by an experiment definition. With the "standard" experiment definition, an event occurs after a creature dies and its genotype's performance is updated. Names for successive autosave files are generated automatically by adding numbers to the last filename you used when saving the EXPT file. For example, when you save an experiment state under the filename **speed1.expt**, and set this parameter to 250, you will get the files **speed1_001.expt**, **speed1_002.expt**, and so on. Each file (log) will be saved after evaluating 250 creatures.

Overwrite: if not set, the simulator will change filenames if needed so that it will not overwrite old files.

Show file comments: comments from loaded files will be displayed in the messages window.

9.3 Error reporting

Choose level of details for messages in the Messages window. Decide when genotypes/creatures are checked against errors.

Fail on warnings: some genotypes cause problems when they construct creatures. They are valid syntactically, but warnings are generated while a creature is built from such a genotype. These warnings are called "build problems", and an example of this situation is when a genotype provides many identical muscles in the same place in the body. This is not allowed, and thus some muscles are disabled (which causes warnings).

If you enable "Fail on warnings", such genotypes will not be simulated, only those that are entirely correct. Otherwise, genotypes with build problems (after they are fixed) will also be simulated.

9.4 World

Type: For 'Blocks' and 'Height field' you have to provide a "map" of heights. 'Height field' world has smooth slopes.

Size: Side length of the (square) world.

Map: description of world heights, it can be:

- randomly generated world: `r (size1) (size2) (seed)`
  size1,size2 are number of blocks in west-east and north-south direction. seed can be omitted, for different numbers you get different worlds.
- custom layout: `m (size1) (size2)`
  followed by world map (size1*size2 digits)
  5 means 'level zero' block height, 6-9 is a 'hill', 1-4 is a 'hole'
  example (a cross-shaped hill):

```
m 3 3
575
777
575
```

additional characters "-" and "|" are slope surfaces between blocks ("-" is WE direction, "|" is NS direction). They are valid only between digits and only for 'Blocks' world type.
more sophisticated example:

```
m 5 5
22-66
222-6
2|333
-6-33
4|498
```

- custom layout: `M (size1) (size2)`
  works just like 'm', but you can use floating point numbers instead of digits (so you can use 5.35, not just 5). Note that with 'M', the value of 0 means 'level zero' height, which is different from 'm' convention.

Water level: the main surface is at 0.0 height and you set the water level here.

Boundaries: what happens when a creature crosses the boundary of the world?

- None – the world is unlimited,
- Fence – creatures cannot go outside,
- Teleport – creatures are moved to the opposite side when trying to cross the border.

9.5 Genetics

Remember history...: the history of genetic operations (which genotype is mutated from which, which genotype was crossed over with which, etc.) is remembered so that you can draw a tree of evolution. Such history can consume large amounts of memory when it contains many (for example, tens of thousands) genotypes.

Genetics: Neurons to add

Checked neurons are those which may be introduced into genotypes during mutations. Note: these settings work for ***f0*** and ***f1*** genotype formats, and not (yet) for ***f4***. ***f4*** uses "NGTS*|@" neuron set.

Genetics: f0

Relative mutation probabilities for f0 genotype language.

Genetics: f1

Settings for f1 genotype language.

Excluded modifiers let you disable using some genotypic symbols during mutations: you can prevent some features from being present in the evolved genotypes. Excluding 'E' and 'e' is needed when energetic efficiency mode is disabled (so that the creatures will not change their starting energy). And, for example, if you don't want your creatures to use different sticks' weights to improve swimming ability in water, exclude 'W' and 'w'.

Proportional crossover: when turned off, random substrings of two parent genotypes are exchanged to form two offspring genotypes (in two-point crossing over). When turned on, cut points in the second parent will be selected proportionally (based on neural genes) to the cutpoints chosen randomly in the first parent. Thus, if both parents have the same number of neurons, then this will be preserved in their children. If, additionally, both parents have identical bodies and neural genes are aligned in both parent genotypes, then their children will also have identical bodies. This may be important if you perform crossover only within species (among similar genotypes – individuals).

Genetics: f1: Morphology contains detailed mutation probabilities concerning physical structure parts in genotypes. **Genetics: f1: Neuron net** contains detailed mutation probabilities concerning neuron net parts in genotypes.

Genetics: f4

Detailed mutation probabilities concerning f4 genotype language. When mutation occurs, it can be either Add node, Delete node, or Modify node. Values expressing probabilities can be adjusted here. Mutation types are shown below.

- Add node
    - Add division
    - Add neural connection
    - Add neural parameter
    - Add repetition
    - Add random symbol
- Delete node
- Modify node

The number of codes which are randomly mutated is also random, varying from 0% to 25% of the genotype genes.

Genetics: Similarity

Determines how the dissimilarity of two genotypes is evaluated. This is important during crossing over, which can depend on the dissimilarity of the two parent genotypes (and can make crossing over of differing genotypes impossible). Similarity can also be involved in fitness reduction for speciation and creating niches. Enter weights for genotype/phenotype criteria concerning 'body' and 'brain'. Refer to scientific papers (a draft from *Theory in Biosciences* **120**, or a LNCS paper) for the idea of dissimilarity computation method (since publication time, the procedure has been updated in minor details). Dissimilarity is also used for clustering of individuals using UPGMA method.

Genetics: Conversions

Specific converters can be enabled and disabled here. They convert various formats of genotypes. Unless you are an experienced user, leave all of them enabled.

9.6 User scripts

These scripts are loaded automatically from the "scripts" subdirectory. Advanced users can create their own scripts.

9.7 Neurons

Neurons: Simulation

Random initialization: if set to zero, neuron states are equal to zero when each individual is created from its genotype and put into the simulator. If you enter a positivie value, neural states are random, and thus NN control has to be more robust (and does not depend on specific initial state values).

Neurons: Active

These settings decide which neurons are simulated in creatures. If a neuron is inactive, its state is initialized when a creature is 'born', and then the state does not change.

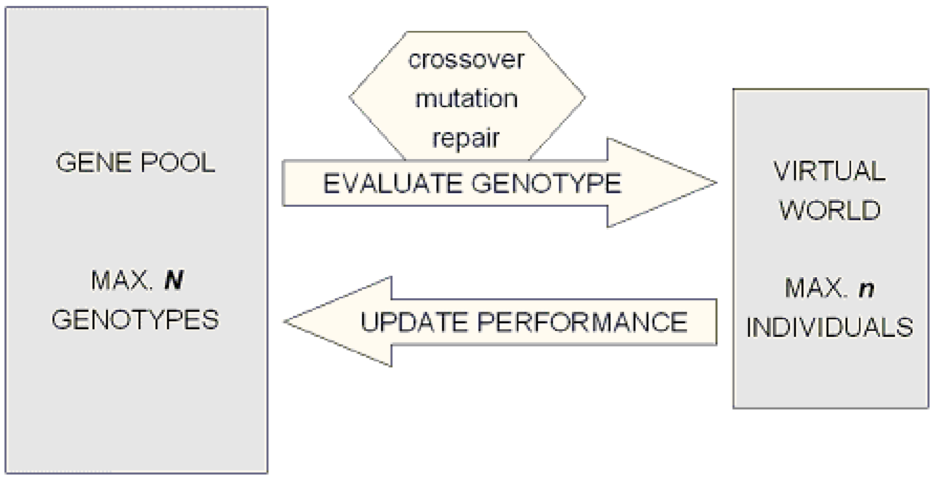
10 "Standard" experiment definition: parameters reference

This is the basic experiment definition, which can be used to perform a range of common experiments. Advanced users can create their own experiment definitions.

Genotypes are stored on the genotypes list. The main idea of this experiment is to evaluate them and create new, probably 'better' ones. Every entry on the genotypes list represents a group of identical genotypes.

There are two main parameters of the system architecture: the capacity of genotypes list (***N***) and the capacity of the virtual world (***n***). The main routine schedules an individual (a genotype) to the world simulator in order to test it (when there is a free space there, it is, when the number of simulated individuals is less than ***n***). After its death the fitness value is calculated (see: Parameters: Fitness).

Note that when ***n***=1, there are no interactions between simulated creatures and the system performs like a typical evolutionary algorithm, where the fitness of an individual does not depend on the other individuals.



"standard.expdef" system architecture.
See animation.

10.1 Tasks performed in each simulation step

- if needed, create new creatures in the world (depends on 'simulated creatures' settings – ***n***)
- if population on the genotypes list exceeds 'capacity' (***N***), some genotypes have to be deleted (see parameters below)
- calculate next step in a 3D world simulation (muscles and neurons: new positions, forces, excitations etc.)
- calculate energy flow
- if some creatures run out of energy, "kill" them and update the genotypes' performance.

Shortcuts for this page:

Parameters

Selection
Fitness
Energy

See also:

General simulation parameters

10.2 Parameters

Initial genotype: the genotype placed in gene pool when the "Initialize experiment" action is called.

Gene pool capacity: the maximal number of genotypes (including copies) in the gene pool.

Delete genotypes: which genotypes will be deleted when capacity limit is reached:

- Randomly
- Inv. proportional fitness (Better fitness = lower chances for deletion)
- Only the worst (remove the one with minimal fitness)

Simulated creatures: the number of creatures automatically put into the simulated world.

Reset performace data: sets "copies" count to zero for each genotype. Thus performace values are meaningless, and first genotype evaluation (and performance averaging, when copies will turn to 1) will override them.

Parameters: Selection

How to choose genotypes for simulation, and how to modify them. Enter how many genotypes relatively will be

- Unchanged – the new genotype is a copy (clone) of the existing (selected) one
- Mutated – the new genotype is a mutation of the existing (selected) one
- Crossed over – the new genotype is a crossover of the existing (selected) two

Minimal similarity: Only the genotypes that are less dissimilar (i.e. more similar) than the given threshold can be crossed-over. This disables breeding of totally different genotypes, whose offspring would probably be inefficient/unable to live. See also Genetics: Similarity.

Parameters: Fitness

These parameters describe how the genotypes' fitness is calculated. Fitness is the weighted sum of criteria shown.

Velocity and Distance are calculated from creature's center of gravity position measured during its lifetime.

Turn on criteria normalization to have the criteria normalized to the interval [0,1] before weighting.

Use similarity speciation to reduce the fitness of genotypes in the gene pool according to their pairwise phenotypic similarity. Note that this operation is time consuming, and requires $n^2/2$ dissimilarity computations every time the gene pool contents changes. See also Genetics: Similarity.

Parameters: Energy

Starting energy: new creature's starting energy (per one stick). This parameter is the base energy value when no energy modifiers ('E'/'e' genes) are used. Such genes modify the actual energy relative to this setting.

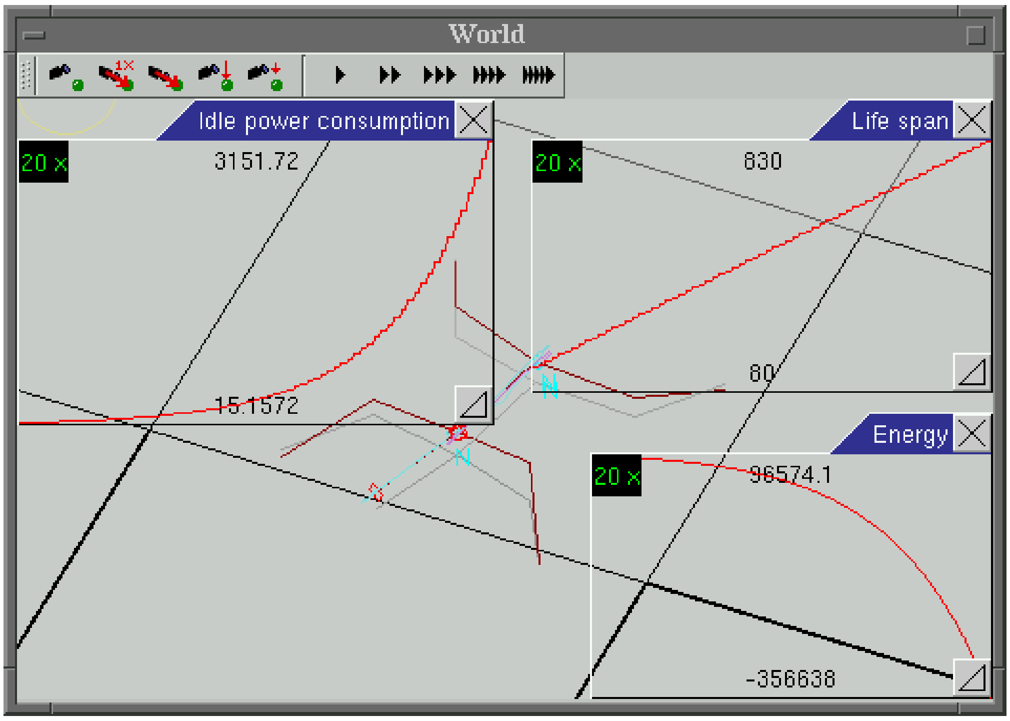
Idle metabolism: energy requirement for one living stick per one simulation step.

Automatic feeding: a given number of energy balls will be placed randomly in the world all the time.

Ball's energy: amount of energy in one ball.

Ingestion multiplier: how fast a creature can ingest energy.

Aging time: When a positive value is set, energy consumption (Idle metabolism) grows non-linearly with life time (see picture below). This setting can be used to stop creatures which ingest more energy than they spend from living forever.



11 Tips on evolution design, parameters, etc.

Framsticks has a powerful, flexible simulator capable of performing various evolutionary processes. Here you will find help concerning setting simulation parameters and general tips on evolutionary process design. The text below concerns the "standard" experiment definition only. Other experiment definitions can be used, extended or developed.

The first question you have to answer is whether you want to simulate a *directed* or a *spontaneous* evolution. If you want to run a *directed* evolution, you will have to explicitly define the fitness criteria used for evaluating of evolved creatures. The simulator will select creatures which are better according to the criteria you choose. You may try to run some kind of a *spontaneous* evolution, where you will have to define the rules of living/survival of virtual creatures, and use life span as the selection criterion. Thus the life span will serve as an estimate of reproductive abilities of creatures. A better realization of a spontaneous evolution is available within the **reproduction.expdef** experiment definition.

11.1 Directed evolution

When performing a directed evolution, the user has to choose the optimization criteria explicitly. This is generally set in the *Experiment* | *Parameters* | *Selection* window. There you set weights for all the criteria. For example, if you set all the weights to zero except "Structure size", the fitness of creatures will correspond exactly to their size. Thus during the evolution bigger creatures will be selected more often than smaller ones. All other fitness criteria of creatures will be ignored.

You can choose many criteria with different weights. You can even use negative numbers. For example, setting "Velocity" to a positive number and "Structure size" to a negative number causes a preference for fast, but small creatures. Of course you have to carefully adjust the actual weight values so that the proper tradeoff is maintained. Setting the "Lifespan" criterion to a non-zero value simulates a spontaneous evolution and you should read the "**Spontaneous evolution**" section (below).

It is important to consider turning on the scaling mechanism, which allows scaling of fitness values in a population. This mechanism normalizes fitness values so that the best genotype is (after scaling) always the same number of times better than the average one. Therefore, better individuals are constantly more preferred during selection, no matter how much better they are. If you are using a tournament selection, consider the tournament size (the bigger the size, the stronger the selection pressure).

In most cases, during a directed evolution effects of interaction of many individuals in the simulated world are not important. Thus you can set "Simulated creatures" to 1 (to evalutate only one creature at a time).

When simulating speed-oriented evolution, you should generally set high values for "Performance sampling period", enable "Performance calculation" after stabilization, set "Starting energy" to 100 or more (to allow a reasonable time of living and speed evaluation after stabilization), and set world "Boundaries" to "None". You can also exclude some characters from the genotype. See "General remarks".

11.2 Spontaneous evolution

With spontaneous evolution, the user has to define more parameters and be more careful. The "Lifespan" criterion should be set to a positive weight. The number of simultaneously "Simulated creatures" should be adjusted in conjunction with the world size, so that not too many of them are simulated, but the interaction is still possible.

"Aging time" should be set to a positive value. You might also set positive values for "Muscle static work", "Muscle dynamic work", "Assimilation energy" and "Automatic feeding", depending on how realistic model you wish to simulate. "Boundaries" may be set to "Teleport" and the world can be land or water.

Remember, the evolution is not directed by any criteria except the life span. The creatures are not evaluated for their specific actions; any behavior that causes prolonged life span is rewarded.

11.3 General remarks

Note that most of the settings described are to be adjusted reasonably, and it is recommended that you know how they work. Being familiar with genetic algorithms and other evolutionary optimization techniques is a great advantage and help.

You should study the meaning of the parameters and anticipate their behavior; it is also good to think over the loop of the evolutionary process simulated in Framsticks and ensure that the settings are rational. Always read hints (tips) about the parameters.

To allow speciation, you can set the "Similarity speciation" and, possibly, the "Genetics: similarity" parameters. See also the "Minimal similarity" parameter.

To simplify the creatures, you may disable some modifiers, receptors and control. You can also disallow undesirable behavior in this way. For example, when simulating creatures on land and evolving them for speed, the S (Smell) receptor is not required. A lot of modifiers (ingestion, assimilation etc.) are also redundant in this situation and should be excluded. Similarly, the T (Touch) receptor is not needed under water (except for touching the bottom, which should most probably not be perceptible by creatures).

Good default values that you should not worry about are those in "Genetics: structure", "Genetics: neuron net" and "Genetics: similarity". "Death" should generally be turned on, unless you are interactively experimenting with the simulation.

A detailed description of all the parameters is available here.

12 Experiment definitions summary

This page summarizes basic experiment definitions that are available within Framsticks. Each section presents one experiment definition along with its properties and short descriptions. All these settings are available in the Framsticks GUI and for scripts.

Jump to experiment definition:

12.1 Experiment definition: Standard Experiment

(short name: standard)

This experiment definition can be used to perform a range of common experiments. It provides

– one gene pool
– one population for individuals
– one "population" for food
– steady–state evolutionary optimization
– fitness as a weighted sum of performance values
– or custom fitness formulas
– fitness scaling
– selection: roulette or tournament
– multiple evaluation option, average and standard deviation available
– can produce logs with average and best fitness
– can detect stagnation and stop automatically
– can save best genotypes

This experiment definition has 39 properties – see table below:

Experiment: Parameters	
Initial genotype	The gene pool will be replaced with the supplied genotype when the experiment begins. Use the empty initial genotype if you want to preserve the current gene pool.
Gene pool capacity	

Delete genotypes	
Simulated creatures	
Initial placement	For 'Central' placement, newborn creatures are placed at the world center, if possible.
Initial orientation	Initial heading of newborn creatures
Initial elevation	Vertical position (above the surface) where newborn creatures are placed. Negative values are only used in the water area: 0 = at the surface –0.5 = half depth –1 = just above the bottom
Clear performance info	Sets the number of instances of each genotype to zero (as if it has never been evaluated). Genotype performance values stay intact, yet they are meaningless if a genotype has no instances.
Experiment: Parameters: Selection	
Unchanged	
Multiple evaluation	If more than zero: – each genotype will be evaluated many times – fitness will be averaged – fitness standard deviation will be stored in the 'user1' field of a genotype – there will be no "Unchanged" genotypes ("Unchanged" value is considered zero).
Mutated	
Crossed over	
Minimal similarity	Only genotypes with dissimilarity below this threshold will be crossed over. Value of 0 means no crossover restrictions.
Selection rule	
Experiment: Parameters: Fitness	
Constant	Constant value added to total fitness
Life span	Weight of life span in total fitness
Velocity	Weight of horizontal velocity in total fitness
Body parts	Weight of body size (number of parts) in total fitness
Body joints	Weight of structure size (number of joints) in total fitness
Brain neurons	Weight of brain size (number of neurons) in total fitness
Brain connections	Weight of brain connections in total fitness
Distance	Weight of distance in total fitness

Vertical position	Weight of vertical position in total fitness
Vertical velocity	Weight of vertical velocity in total fitness
Criteria normalization	Normalize each criterion to 0..1 interval before weighting
Similarity speciation	If enabled, fitness of each genotype will be reduced by its phenotypic similarity to all other genotypes in the gene pool
Experiment: Parameters: Energy	
Starting energy	Base starting energy level (for each stick)
Idle metabolism	Each stick consumes this amount of energy in one time step
Automatic feeding	Number of energy pieces in the world
Food's energy	
Food's genotype	The default food model is a simple, single part object: //0 m:Vstyle=food p: (this genotype is used when you leave this field blank). You can specify another genotype to create "intelligent" or mobile food.
Ingestion multiplier	
Aging time	Idle metabolism doubles after this period (0 disables aging)
Experiment: Parameters: Extras	
Stagnation (auto stop)	No improvement period required to stop simulation (stagnation). 0 disables automatic stopping.
Minimal fitness	Minimal fitness that allows to automatically stop evolution when stagnation detected
Boost phase after stagnation	After stagnation has been detected, switches negative selection to "delete worst", doubles "multiple evaluation" and starts another stagnation detection phase. "Delete worst" results in extremely quick convergence and high selection pressure, similarly to "local search" optimization techniques.
Sound on improvement	Emit sounds on improvements? (frequency depends on the magnitude of improvement)
Save improvements	Saves (on each improvement) best found genotype or a complete experiment state to a file in scripts_output subdirectory.
Log fitness	Sends [LOG] messages with genotypes count and minimal, average and best gene pool fitness, which can be used to produce graphs by external tools like gnuplot or Excel. It also sends the [LOGTITLE] message on experiment initialization, which summarizes most important parameters of your experiment. It can be used as a graph title.

12.2 Experiment definition: Asexual reproduction in the world

(short name: reproduction)

Spontaneous evolution. Each creature with a sufficient energy level produces an offspring, which is then put close to its parent.
Food is created at a constant rate and placed randomly.

This experiment definition has 11 properties – see table below:

Experiment: Parameters	
Initial genotype	
Creation height	Vertical position (above the surface) where new creatures are revived. Negative values are only used in the water area: 0 = at the surface –0.5 = half depth –1 = just above the bottom
Mutations	
Restart after extinction	Restart automatically this experiment after the last creature has died?
Experiment: Parameters: Energy	
Starting energy	Initial energy for the first creature
Reproduction energy	Creature can produce offspring when its energy level reaches this threshold
Idle metabolism	Each stick consumes this amount of energy in one time step
Feeding rate	How fast energy is created in the world
Food's energy	
Food's genotype	The default food model is a simple, single part object: //0 m:Vstyle=food p: (this genotype is used when you leave this field blank). You can specify another genotype to create "intelligent" or mobile food.
Ingestion multiplier	

12.3 Experiment definition: Demonstration of evolution

(short name: evolution_demo)

This experiment definition demonstrates the process of evolution.
Individuals are placed in a circle. A new individual is cloned, mutated or crossed over.
Then it is evaluated in the middle of the circle, and, depending on its fitness, may replace a poorer, existing individual or disappear.

This experiment definition has 19 properties – see table below:

Experiment: Parameters	
Initial genotype	Initial genotype, mutated to create initial population of genotypes. You need to (re)initialize the experiment to (re)create initial population of genotypes.
Delete genotypes	
Simulated creatures	You need to (re)initialize the experiment to change this setting and create initial population of given size.
Initial orientation	Initial heading of newborn creatures
Initial elevation	Vertical position (above the surface) where newborn creatures are placed. Negative values are only used in the water area: 0 = at the surface –0.5 = half depth –1 = just above the bottom
Predefined setup	
Number of steps each creature lives	Number of simulation steps for the creature in the center of cratures circle. Beware to performance sampling period when setting this value.
Number of idle visualization steps	Number of idle visualization steps: – after parent (parents) approaches (approach) the center and before new creature is born in the world – after new creature is born in the world and before parent (parents) returns (return) to original place (places) – after central creature disappearance – after replacement of circle circumference creature by central creature If this parameter is set to zero, than simulation speeds up. If this parameter is set to big value, then some simulation details may be better observed. It is possible to change this value during simulation, to obtain different goals.
Experiment: Parameters: Selection	
Unchanged	

Mutated	
Crossed over	
Minimal similarity	Only genotypes with dissimilarity below this threshold will be crossed over. Value of 0 means no crossover restrictions.
Selection rule	
Experiment: Parameters: Fitness	
Constant	Constant value added to total fitness
Vertical position	
Velocity	Weight of horizontal velocity in total fitness
Body parts	Weight of body size (number of parts) in total fitness
Criteria normalization	Normalize each criterion to 0..1 interval before weighting
Similarity speciation	If enabled, fitness of each genotype will be reduced by its phenotypic similarity to all other genotypes in the gene pool

12.4 Experiment definition: Neuroanalysis

(short name: neuroanalysis)

This experiment evaluates all genotypes in the gene pool.
During simulation, the output signal of each neuron is
analyzed, and its average and standard deviation
are computed. These data are then saved in the 'Info'
field of the genotype.

This experiment definition has 3 properties – see table below:

Experiment: Parameters	
Evaluation time	
Restart from the first genotype	
Initial elevation	Vertical position (above the surface) where newborn creatures are placed. Negative values are only used in the water area: 0 = at the surface –0.5 = half depth –1 = just above the bottom

12.5 Experiment definition: Learn where food is, explore and exploit

(short name: learn_food)

When an individual encounters food, it eats a bit of it and remembers its location.
It also gets "refreshed" (i.e. gets a full amount of energy). Energy of each
individual provides information on how recent (current) is the food location
information that the individual stores. Information recency is visualized as
brightness of individuals (light green ones have recently found food).
When individuals collide, they learn from each other where food is (by averaging their knowledge).
A newborn individual moves randomly and receives (duplicates) knowledge from the
first knowledge–rich individual that it collides with.
An individual that cannot find food for a long period of time dies, and a newborn
one is created.

–=–

An interesting phenomenon to be observed in this experiment is how sharing information
helps explore food location. Food items can be added either close to previous items,
or randomly (in the latter case, the information about food location is not very useful
for individuals). You can turn off automatic feeding and keep adding food manually to see
how learning influences behavior of the population. See the "share knowledge" parameter (on/off).

With learning, individuals do not have to find food themselves.
They can also get in contact with other individuals that know
where the food was, and exchange information (the values learned
are proportional to the recency of information). It is interesting
to see how knowledge sharing (cooperation, dependence) versus
no sharing (being self–sufficient, independence, risk) influences minimal,
average and maximal life span in the neighboring and random food placement scenarios.

Notions of exploration of the environment and exploitation of knowledge
about the environment are illustrated as well in this experiment.

–=–

The dynamics of this system depends on the following parameters:
– number of individuals and world size
– size and shape of their body (affects collisions and sharing of knowledge)
– food eating rate
– food placement (neighboring or random)
– learning strategy (e.g. weighted averaging of food coordinates)
– behavior (e.g. move within circles, small after finding food, then larger and larger)

–=–

Technical:
Food location (x,y) is stored in user1,2 fields of each individual.

This experiment definition has 11 properties – see table below:

Experiment: Parameters	
Number of creatures	
Share knowledge	Share knowledge about food position when two creatures collide?
Initial genotype	
Creation height	Vertical position (above the surface) where new creatures are revived. Negative values are only used in the water area: 0 = at the surface –0.5 = half depth –1 = just above the bottom
Experiment: Parameters: Energy	
Starting energy	Initial energy for the first creature
Idle metabolism	Each stick consumes this amount of energy in one time step
Feeding rate	How fast energy is created in the world
Food's energy	
Food placement	Random placement contradicts 'learning food location' and therefore constitutes a test/benchmark for this experiment definition.
Food's genotype	The default food model is a simple, single part object: //0 m:Vstyle=food p: (this genotype is used when you leave this field blank). You can specify another genotype to create "intelligent" or mobile food.
Ingestion multiplier	

12.6 Experiment definition: Boids

(short name: boids)

Boids, developed by Craig Reynolds in 1986, is an artificial life program, simulating the flocking behaviour of birds.

As with most artificial life simulations, Boids is an example of emergent behaviour; that is, the complexity of Boids arises from the interaction of individual agents (the boids, in this case) adhering to a set of simple rules. The rules applied in the simplest Boids world are as follows:

* separation: steer to avoid crowding local flockmates
* alignment: steer towards the average heading of local flockmates
* cohesion: steer to move toward the average position of local flockmates

(http://en.wikipedia.org/wiki/Boids)

This experiment definition has 6 properties – see table below:

Experiment: Parameters
Number of boids
Rule 1 - Separation
Rule 2 - Alignment
Rule 3 - Cohesion
Neighborhood range
Separation distance

12.7 Experiment definition: Generational evolutionary optimization experiment

(short name: generational)

A simple "genetic algorithm" experiment:

– two gene pools (previous and current generation)
– one population for individuals
– generational replacement of genotypes
– selection: roulette (fitness–proportional)
– fitness formula entered directly into the group's field

This experiment definition has 8 properties – see table below:

Experiment: Parameters	
Initial genotype	
Gene pool size	
Evaluation time	
Restart epoch	Re–evaluate all genotypes in the current generation
Initial elevation	Vertical position (above the surface) where newborn creatures are placed. Negative values are only used in the water area: 0 = at the surface –0.5 = half depth –1 = just above the bottom

Experiment: Parameters: Selection
Unchanged
Mutated
Crossed over

12.8 Experiment definition: Mazes

(short name: mazes)

This experiment definition can be used to evaluate (and evolve)
creatures moving between two specified points in a maze. These points
are indicated by start and target marks (in the second population).

Genotype's user1 field (which acts as a maximized fitness) contains
– energy left (when target found during lifespan)
– distance to the target (as a negative value)
when a creature died away from the target.

Press "SHIFT" and click the right mouse button to manually set
the start and finish points.

Use maze[1,2].sim settings for this experiment definition.

This experiment definition has 39 properties – see table below:

Experiment: Parameters	
Initial genotype	The gene pool will be replaced with the supplied genotype when the experiment begins. Use the empty initial genotype if you want to preserve the current gene pool.
Gene pool capacity	
Delete genotypes	
Simulated creatures	
Initial placement	For 'Central' placement, newborn creatures are placed at the world center, if possible.
Initial orientation	Initial heading of newborn creatures
Initial elevation	Vertical position (above the surface) where newborn creatures are placed. Negative values are only used in the water area: 0 = at the surface –0.5 = half depth –1 = just above the bottom

Start X positions	A comma–separated list of X positions (based on the world map) of starting points
Start Y positions	A comma–separated list of Y positions (based on the world map) of starting points
Start headings	A comma–separated list of initial headings (in degrees)
Target X positions	A comma–separated list of X positions (based on the world map) of target points
Target Y positions	A comma–separated list of Y positions (based on the world map) of target points
Target radius	The target is a circle with the radius defined here
Clear performance info	Sets the number of instances of each genotype to zero (as if it has never been evaluated). Genotype performance values stay intact, yet they are meaningless if a genotype has no instances.
Experiment: Parameters: Selection	
Unchanged	
Multiple evaluation	If more than zero: – each genotype will be evaluated many times – fitness will be averaged – fitness standard deviation will be stored in the 'user1' field of a genotype – there will be no "Unchanged" genotypes ("Unchanged" value is considered zero).
Mutated	
Crossed over	
Minimal similarity	Only genotypes with dissimilarity below this threshold will be crossed over. Value of 0 means no crossover restrictions.
Selection rule	
Experiment: Parameters: Fitness	
Constant	Constant value added to total fitness
Life span	Weight of life span in total fitness
Velocity	Weight of horizontal velocity in total fitness
Body parts	Weight of body size (number of parts) in total fitness
Body joints	Weight of structure size (number of joints) in total fitness
Brain neurons	Weight of brain size (number of neurons) in total fitness
Brain connections	Weight of brain connections in total fitness

Distance	Weight of distance in total fitness
Vertical position	Weight of vertical position in total fitness
Vertical velocity	Weight of vertical velocity in total fitness
Criteria normalization	Normalize each criterion to 0..1 interval before weighting
Similarity speciation	If enabled, fitness of each genotype will be reduced by its phenotypic similarity to all other genotypes in the gene pool
Experiment: Parameters: Energy	
Starting energy	Base starting energy level (for each stick)
Idle metabolism	Each stick consumes this amount of energy in one time step
Automatic feeding	Number of energy pieces in the world
Food's energy	
Food's genotype	The default food model is a simple, single part object: //0 m:Vstyle=food p: (this genotype is used when you leave this field blank). You can specify another genotype to create "intelligent" or mobile food.
Ingestion multiplier	
Aging time	Idle metabolism doubles after this period (0 disables aging)

12.9 Experiment definition: Text writer

(short name: text_writer)

This experiment definition displays formatted (and flowing) text using
creatures as letters and digits. It requires the "fonts.gen" file.
Only big letters and digits can be used in the text.

Initialize the experiment to situate the text.

The text can be formatted using HTML–like tags:

<left>
<center>
<right>
<justify> – are used to start a paragraph

<hNUMBER> – sets the height (elevation) of the text (see also the 'Gravity' setting)

<hsNUMBER> – sets hotizontal spacing between letters

<vsNUMBER> – sets vertical spacing between lines

<f0>
<f1> – select the font (f0 is more regular)

This experiment definition has 3 properties – see table below:

Experiment: Parameters	
Text	The text you want to see
Collisions	When turned on, nearby letters will collide and bounce
Speed	Text movement: 1–slowest, 100–fastest

12.10 Experiment definition: Dance

(short name: dance)

A synchronous framsdance :–)

This experiment definition has 38 properties – see table below:

Experiment: Parameters	
Initial genotype	The gene pool will be replaced with the supplied genotype when the experiment begins. Use the empty initial genotype if you want to preserve the current gene pool.
Gene pool capacity	
Delete genotypes	
Simulated creatures	
Initial placement	For 'Central' placement, newborn creatures are placed at the world center, if possible.
Initial orientation	Initial heading of newborn creatures
Initial elevation	Vertical position (above the surface) where newborn creatures are placed. Negative values are only used in the water area: 0 = at the surface –0.5 = half depth –1 = just above the bottom
Land dance	
Water dance	
Number of dancers	

Dance tempo	
Change arrangement	
Beat	
Experiment: Parameters: Selection	
Unchanged	
Multiple evaluation	If more than zero: – each genotype will be evaluated many times – fitness will be averaged – fitness standard deviation will be stored in the 'user1' field of a genotype – there will be no "Unchanged" genotypes ("Unchanged" value is considered zero).
Mutated	
Crossed over	
Minimal similarity	Only genotypes with dissimilarity below this threshold will be crossed over. Value of 0 means no crossover restrictions.
Selection rule	
Experiment: Parameters: Fitness	
Constant	Constant value added to total fitness
Life span	Weight of life span in total fitness
Velocity	Weight of horizontal velocity in total fitness
Body parts	Weight of body size (number of parts) in total fitness
Body joints	Weight of structure size (number of joints) in total fitness
Brain neurons	Weight of brain size (number of neurons) in total fitness
Brain connections	Weight of brain connections in total fitness
Distance	Weight of distance in total fitness
Vertical position	Weight of vertical position in total fitness
Vertical velocity	Weight of vertical velocity in total fitness
Criteria normalization	Normalize each criterion to 0..1 interval before weighting
Similarity speciation	If enabled, fitness of each genotype will be reduced by its phenotypic similarity to all other genotypes in the gene pool
Experiment: Parameters: Energy	
Starting energy	Base starting energy level (for each stick)

Idle metabolism	Each stick consumes this amount of energy in one time step
Automatic feeding	Number of energy pieces in the world
Food's energy	
Food's genotype	The default food model is a simple, single part object: //0 m:Vstyle=food p: (this genotype is used when you leave this field blank). You can specify another genotype to create "intelligent" or mobile food.
Ingestion multiplier	
Aging time	Idle metabolism doubles after this period (0 disables aging)

12.11 Experiment definition: Framsbots - Game

(short name: framsbots)

Framsbots is a simple game inspired by the classic Robots game.
The aim of this game is to run away from hostile creatures and make all of them hit each other.
Just click somewhere (left–click or double–right–click) to move your creature (the one that is in the middle of the world in the beggining). Your creature will go towards the point you clicked.
All the enemies will move towards you. Use this information to make them hit each other, so they will loose energy and die.
If you see an apple, try to get it. You will gain energy and you may even get a new life!
Use shift+left mouse drag to rotate world.

Read more about this game:
http://www.framsticks.com/wiki/FramsBots

This experiment definition has 16 properties – see table below:

Experiment: Parameters	
Enemy creature type	
Player creature type	
Level	Number of a level to play (–1 is random)
Show additional debug messages	
Experiment: Parameters: Enemies	
Number of enemy creatures	
Starting energy of enemy creature	Base starting energy level

Multiplier of energy taken by Enemy	
Multiplier of energy taken by frozen Enemy	
Multiplier of energy taken when Enemies collide	
Multiplier of energy taken from alone Enemy	
What to do when Enemies die	
Experiment: Parameters: Player	
Starting energy of player creature	Base starting energy level
Experiment: Parameters: Food	
Starting energy of food	Base starting energy level
Amount of energy lost	How much energy food looses each step
Food probablity	Probability of food appearing after enemy killed
Experiment: Parameters: Azimuth	
Maximum length of positions history vectors	

12.12 Experiment definition: Batch evaluation of loaded genotypes

(short name: standard-eval)

Use this experiment to evaluate all genotypes one by one.
Use gene pool capacity parameter to set the required number of evaluations of each genotype.

The genotypes for evaluation _must_ be different.

First load your genotypes for evaluation, then initialize experiment,
then adjust all simulation parameters, and finally run the simulation
to perform all evaluations.

After evaluation, fitness of each genotype contains the average fitness,
user1 field contains standard deviation, and user2 field contains the
average time (in seconds) needed for single evaluation.

This experiment definition has 33 properties – see table below:

Experiment: Parameters	
Initial genotype	The gene pool will be replaced with the supplied genotype when the experiment begins. Use the empty initial genotype if you want to preserve the current gene pool.
Gene pool capacity	

Delete genotypes	
Simulated creatures	
Initial placement	For 'Central' placement, newborn creatures are placed at the world center, if possible.
Initial orientation	Initial heading of newborn creatures
Initial elevation	Vertical position (above the surface) where newborn creatures are placed. Negative values are only used in the water area: 0 = at the surface –0.5 = half depth –1 = just above the bottom
Clear performance info	Sets the number of instances of each genotype to zero (as if it has never been evaluated). Genotype performance values stay intact, yet they are meaningless if a genotype has no instances.
Experiment: Parameters: Selection	
Unchanged	
Multiple evaluation	If more than zero: – each genotype will be evaluated many times – fitness will be averaged – fitness standard deviation will be stored in the 'user1' field of a genotype – there will be no "Unchanged" genotypes ("Unchanged" value is considered zero).
Mutated	
Crossed over	
Minimal similarity	Only genotypes with dissimilarity below this threshold will be crossed over. Value of 0 means no crossover restrictions.
Selection rule	
Experiment: Parameters: Fitness	
Constant	Constant value added to total fitness
Life span	Weight of life span in total fitness
Velocity	Weight of horizontal velocity in total fitness
Body parts	Weight of body size (number of parts) in total fitness
Body joints	Weight of structure size (number of joints) in total fitness
Brain neurons	Weight of brain size (number of neurons) in total fitness
Brain connections	Weight of brain connections in total fitness
Distance	Weight of distance in total fitness

Vertical position	Weight of vertical position in total fitness
Vertical velocity	Weight of vertical velocity in total fitness
Criteria normalization	Normalize each criterion to 0..1 interval before weighting
Similarity speciation	If enabled, fitness of each genotype will be reduced by its phenotypic similarity to all other genotypes in the gene pool
Experiment: Parameters: Energy	
Starting energy	Base starting energy level (for each stick)
Idle metabolism	Each stick consumes this amount of energy in one time step
Automatic feeding	Number of energy pieces in the world
Food's energy	
Food's genotype	The default food model is a simple, single part object: //0 m:Vstyle=food p: (this genotype is used when you leave this field blank). You can specify another genotype to create "intelligent" or mobile food.
Ingestion multiplier	
Aging time	Idle metabolism doubles after this period (0 disables aging)

13 Graphical User Interface (GUI) for Windows

Other interfaces:

- command line (all platforms)
- Amiga GUI
- network server

See also: Interface parameters, Simulator parameters
Bottom: Mini-tutorial, GUI command-line usage

13.1 Introduction

Click on the "Window" menu. Notice that there are seven windows to show/hide. By default, five of them are already open: Groups, Artificial world, Body & Brain, and two lists: genotypes and individuals. Genotypes and individuals (or, generally, simulated objects) can be grouped into Gene pools and Populations, respectively.

You can revive individuals by selecting their genotypes (upper list) and pressing the "Simulate" button. The selected genotypes are then "grown" and placed in the artificial world. Selecting them on the "Populations" list makes the camera follow the selected creatures (display rate is adjustable in the listbox in the top right corner of the world window).

Use the menu to start and stop the simulation. You can also invoke some functions using the toolbar on the top, or using keyboard shortcuts.

You can kill or delete living creatures using buttons. Killing means that the program treats the creature as if its energy had reached zero (so it is like a "natural" death). Deleting means just removing it from simulation.

Update listbox (in the "Populations" window) sets the frequency for refreshing information about genotypes, creatures, etc. When you double-click a genotype on the list, the genotype data window will be opened:

13.2 Genotype data window

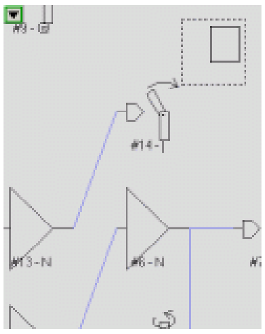
You can see/edit properties of genotypes. While editing the genotype, you can interactively see the changes (instant body and brain preview), as long as the genotype is valid. Selecting a genotype fragment highlights parts of the morphology and brain which are created by the fragment. It also works the other way: you can select body or brain parts to see what genes are responsible for creating these parts (these genes are underlined). Hold down the Shift key while selecting parts of body and brain to add new parts to the selection, or to toggle them.

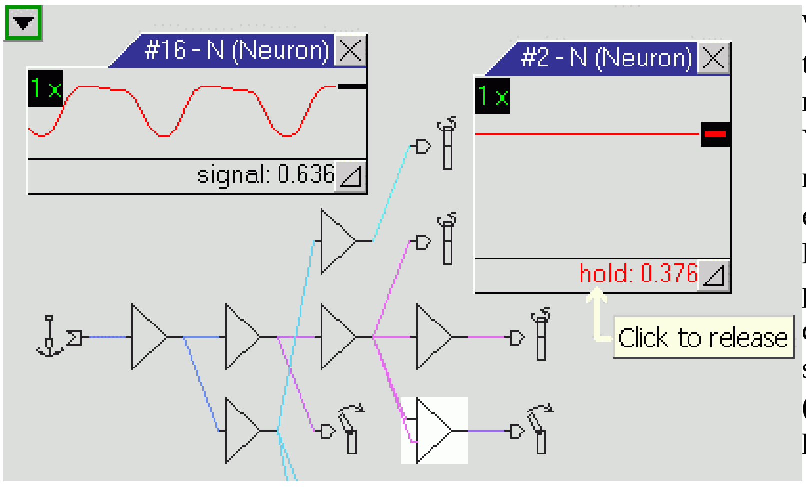
You can give any name to your creature ('Name' field). If you leave it blank, the name will be generated automatically.

13.3 Body and brain window

In the body panel, left click selects parts of the body or neurons/receptors/effectors. Hold down Shift to add more parts to the selection. Right mouse drag rotates the structure. The mouse wheel zooms.

In the brain panel, left click selects parts of the brain: neurons, receptors, and effectors. Hold down Shift to add more parts to the selection. The mouse wheel zooms. You can move neurons by draging them while you hold the Shift key down. Press the Alt key and drag the mouse horizontally to zoom the neural diagram (as shown on the right picture). When zoomed, you can drag the diagram to pan it.





When the neural network shown belongs to a living creature, double clicking on a neuron makes a signal window pop up. You can not only see signal flow in neurons, but also adjust levels of excitations and thus control the brain of a living creature (as shown on the left picture). Also, neural connections are displayed in various colors reflecting the signal level. Click on the scale indicator (the green "1x") and drag the mouse horizontally to scale the chart.

Helpful hints are displayed when you stop the mouse pointer over an object.

13.4 Artificial world window

In this window, left mouse drag rotates the world. Right mouse drag pans the view, and mouse wheel zooms. Double clicking on a simulated object will focus the camera on that object (although this will not be reflected by a selection of that object on the populations list).

Press Shift + right mouse click to invoke an action from the action list. Right double click will invoke the first action from the list without displaying it.

Press Control + left mouse click to grab some object with a robot arm and move it by dragging the mouse. Release the left mouse button first to drop the object. Release the Control key first to leave the robot arm holding the object.

13.5 More hints

Be sure to explore the menus which appear when you click on a small arrow in top left corner of windows. Also, right click on genotypes and individuals which are listed in gene pools and populations to see many options.

Messages produced by the program can be viewed in the "Messages" window.

13.6 A GUI mini-tutorial

1. Launch the Framsticks application
2. Load genotypes from *walking.gen* (Menu:File->Load genotypes from...)
3. Load parameters from *manual.sim* (Menu:File->Load simulator parameters from...)
4. Select 'Basic Quadruped' on the Genotypes (upper) list
5. Click the **Simulate** button and choose the "Creatures" population
6. Select 'Basic Quadruped' on the Individuals (lower) list
7. Start simulation (Menu:Simulation->Run)
8. You can watch as the creature moves, play with it, grab with the mouse (control + left mouse button), move the camera (left or right drag on the world window), zoom (mouse wheel), etc.
9. Double-click on the 'Basic Quadruped' in the upper window to examine its data.
10. Want to try out your own creature? Look at the 'genotype' section, and try to modify it or create a new one. Then you can place it in the simulator world.

If you would like to start some evolution now, you should change some parameters: turn on 'death' – creatures will be removed when they run out of energy, set the number of creatures to be placed automatically in the world, set the selection criterion... and of course enter the starting genotype in the genotypes list: this can be the simple 'bacteria' (single X) or you may wish to improve any other genotype by evolution.

13.7 Command-line parameters for the Framsticks GUI

Framsticks.exe [-go | -demo] [-help] [file.sim] [file.gen] [...]

Examples:

Framsticks.exe ev11_11.gen ev11_12.gen

Framsticks.exe water.sim ev.gen -go

Framsticks.exe -demo demo-chase.sim demo-chase.gen

Framsticks.exe manual.sim

The first example loads two genotype files.
The second example loads a water environment and a genotype file, and starts evolution.
The third example loads an environment and genotypes, and runs the demo mode (80% displaying time, 20% pure simulation).
The fourth example loads a simulator settings file.

13.8 Advanced notes

- Simulator settings files may contain partial information (only a subset of parameters). That may be very useful when defining various environments and batch processing.
- You can drag **.gen*, **.sim*, and **.expt* files and drop them on *Framsticks.exe* file in order to launch Framsticks and load the dragged files. You can also double-click files with such extensions to launch Framsticks and load the clicked file.

14 Interface parameters

If you need help, read the hints (tooltips) which appear when you stop the mouse pointer over the name of a parameter in the program. Documentation below is provided in addition to those hints and discusses only some issues.

See also:

Simulation Parameters
Interface

This page:

14.1 General

Save on exit: save all settings from this window when you exit Framsticks?

Demo mode: allows you to leave your computer evolving, while occasionally displaing simulated creatures. However, display slows down simulation, so the less % of time you choose, the faster the evolution will proceed. Using "demo mode" is good for demonstrations and presentations of Framsticks.

Keep extensions associated: should be turned on if you want Framsticks file extensions to be associated with this program. This allows you to open such files by double-clicking them, and also displays appropriate icons for such files. When this option is turned on, Framsticks ensures that these associations are valid each time you run the program.

14.2 Visual style

Style: registered users (full license) can choose from a list of visual styles for OpenGL view. Advanced users can create their own styles and themes with their own textures and settings. Source code for styles is in the "3dobj" subdirectory.

14.3 Performance charts

Simulator charts

Performance charts concern living (currently simulated) creatures. You have to select a creature to see its performance on the chart.

Simulator charts concern genetics, gene pools and populations.

Selected property: choose a property (criterion) to add to the Charts window. After you select it, press "Apply" and then the "Add chart" button.

14.4 Export world

Export world: POV-Ray

Registered users with a "full" license can export scenes from the simulated world in POV-Ray format. POV-Ray is a powerful, freeware raytracer. To render a scene, you have to install POV-Ray. To make a movie, you need additional software which joins individual frames rendered by POV-Ray. You can get more information at the official POV-Ray site, www.povray.org.

Framsticks can create scene files for use with POV-Ray. It gives you the possibility to view photorealistic pictures and movies from your creatures' artificial world. To render such pictures, POV-Ray needs the "framsticks.inc" include file, so you have to make it accessible (e.g. copy it to your POV-Ray include directory). This file is very important, as it contains the definitions of all the rendered objects – sticks, muscles, receptors, water, fence, hills, etc. If you know POV-Ray, you can even modify this file to obtain different Framsticks scenarios.

After the export has been enabled, the simulator creates the file "world.inc" in the selected directory (this file contains all the static objects in the simulated world: ground, sky, light etc.). Then, an exported scene file is created in each simulation step (but observing the skip frames parameter). Every scene file #includes "world.inc". If POV-Ray has trouble accessing "world.inc", copy this file to the directory where .pov scene files are created.

The meaning of POV-Ray export parameters is analogous to those of 'Other formats' section (see below). **scene_%04d.pov** is the default output files pattern for POV-Ray export.

Export world: Other formats

Registered users with a "full" license can export scenes from the simulated world in a range of formats. However, exported files are often simplified, and some formats on the list may be not available.

Output files pattern: when the scene export is enabled, the program creates exported scene files. Names for successive scene files are generated automatically. For example, when your output files pattern is **frame_%05d.ac**, you will get the files **frame_00001.ac**, **frame_00002.ac**, and so on ('5' in the pattern means five digits, '0' means zero-padding). If you set the pattern to **img%d.ssg**, you will get the files **img1.ssg**, **img2.ssg**, and so on ('%d' is simply replaced by consecutive numbers without any special formatting).

Flatten scene: joins all world objects into one object. This is required for most export formats, otherwise you may only get a single object for each object type.

Enable export: turns on exporting mode, where an exported scene file is created in each simulation step (but observing the skip frames parameter). To check whether the program is in exporting mode, see the Export enabled checkbox.

Export world: OpenGL image

When the OpenGL world view is activated, bitmaps from the world window may be exported as image files.

15 OpenGL visualisation

OpenGL is an industrial standard for 3D graphics (see www.opengl.org). It is becoming more and more popular; the development of new 3D graphics accelerators and faster processors available for everyone makes it an important and useful standard.

Framsticks OpenGL gives a very impressive three-dimensional view of the virtual world and body structures of the creatures. Creatures' sticks are displayed as solid cylinders with textures, and all objects are shaded (almost like renderings). All this is possible in real time.

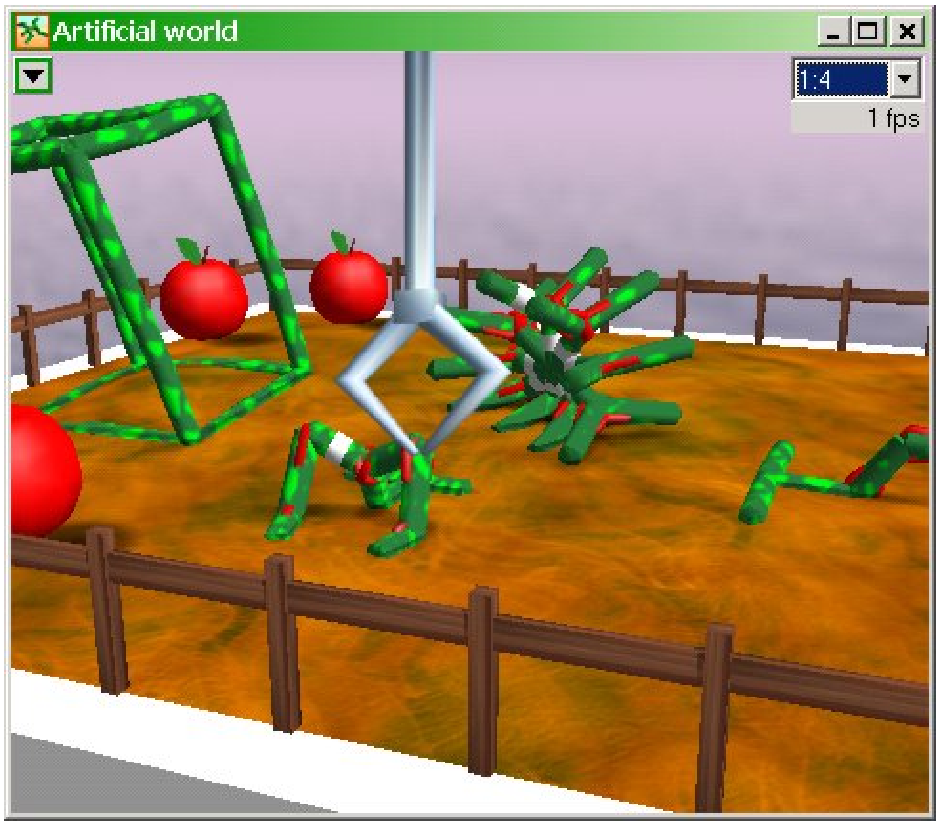
Click on the picture to get a full-sized view. Download a **free** program, FramsView (for Windows or Linux) to see 3D OpenGL Framsticks structures. You can also download the trialware Framsticks Theater (for Windows or Linux) to see real time, 3D OpenGL view of various simulations.

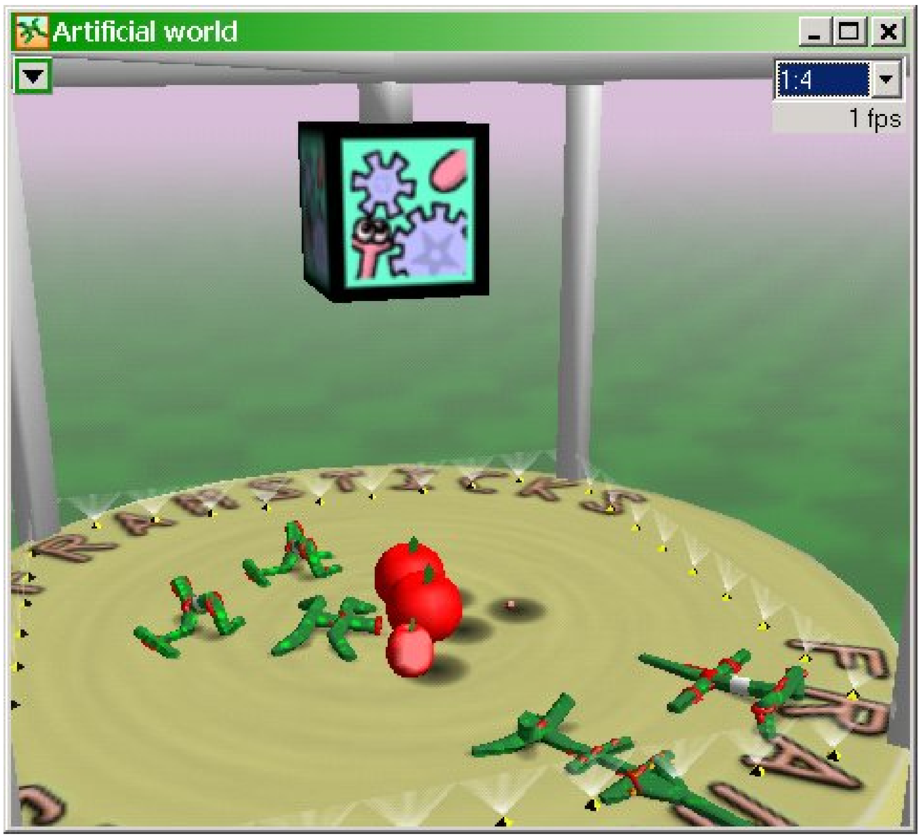
15.1 Some OpenGL styles

(advanced users can create their own styles!)

Standard

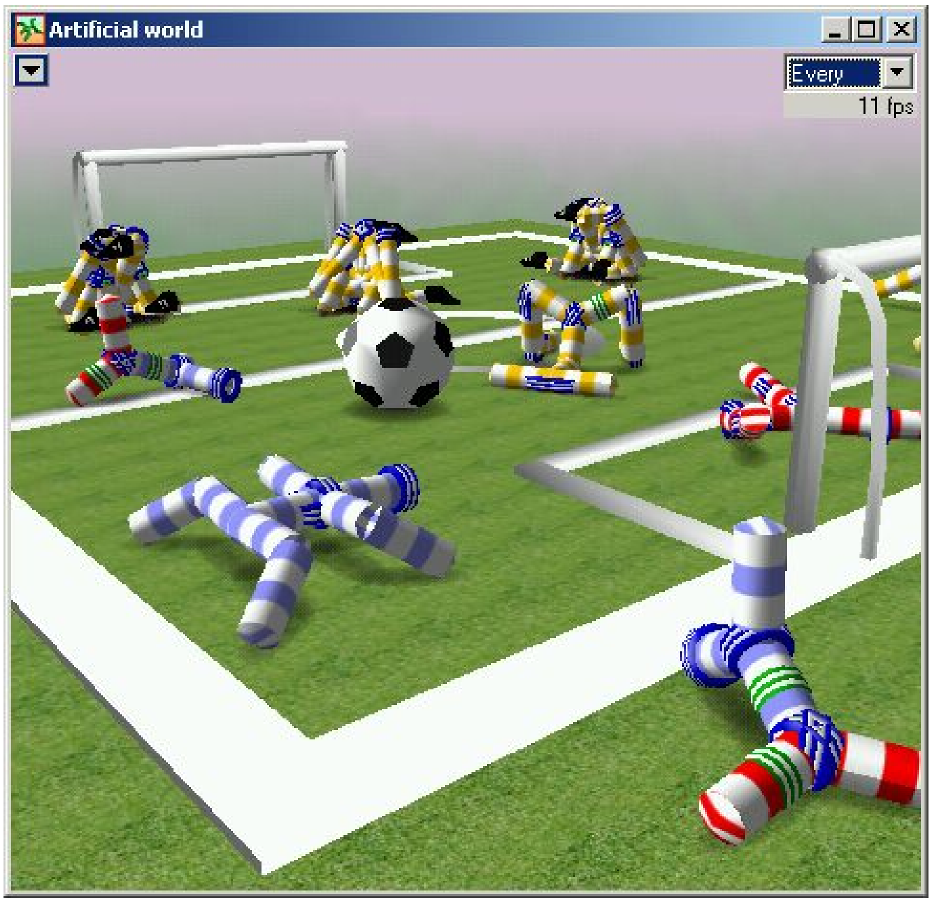
(default Framsticks visual style)





Arena

(a round arena)

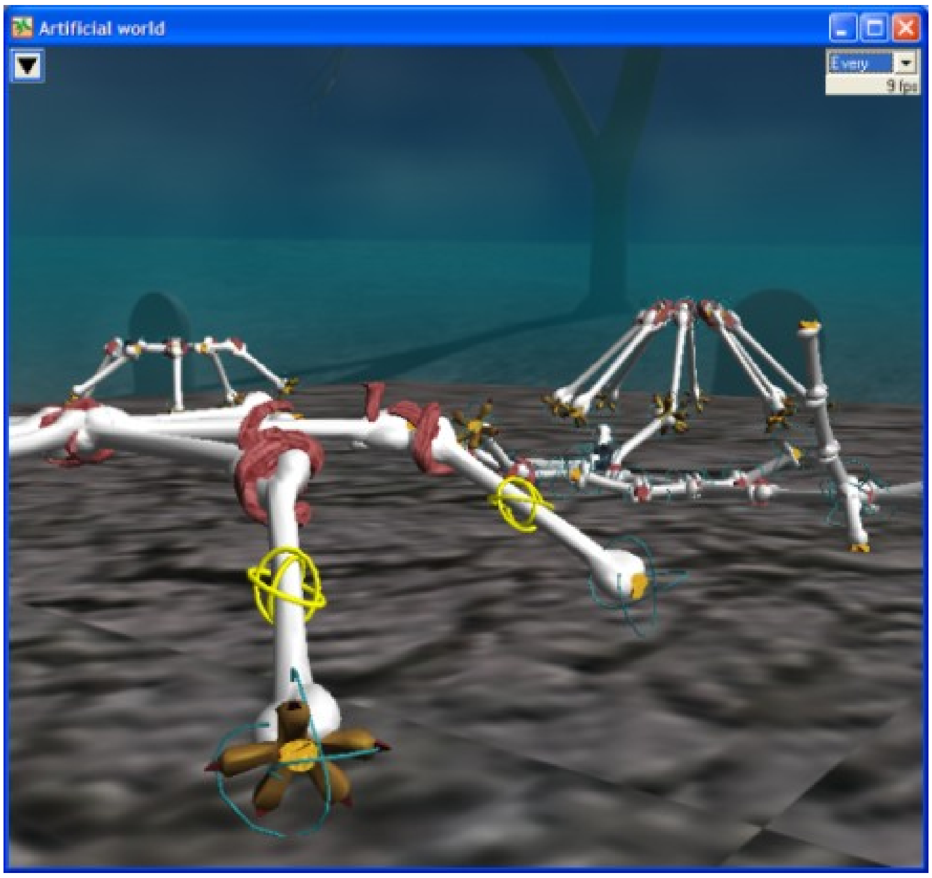


Football

(football players and balls)

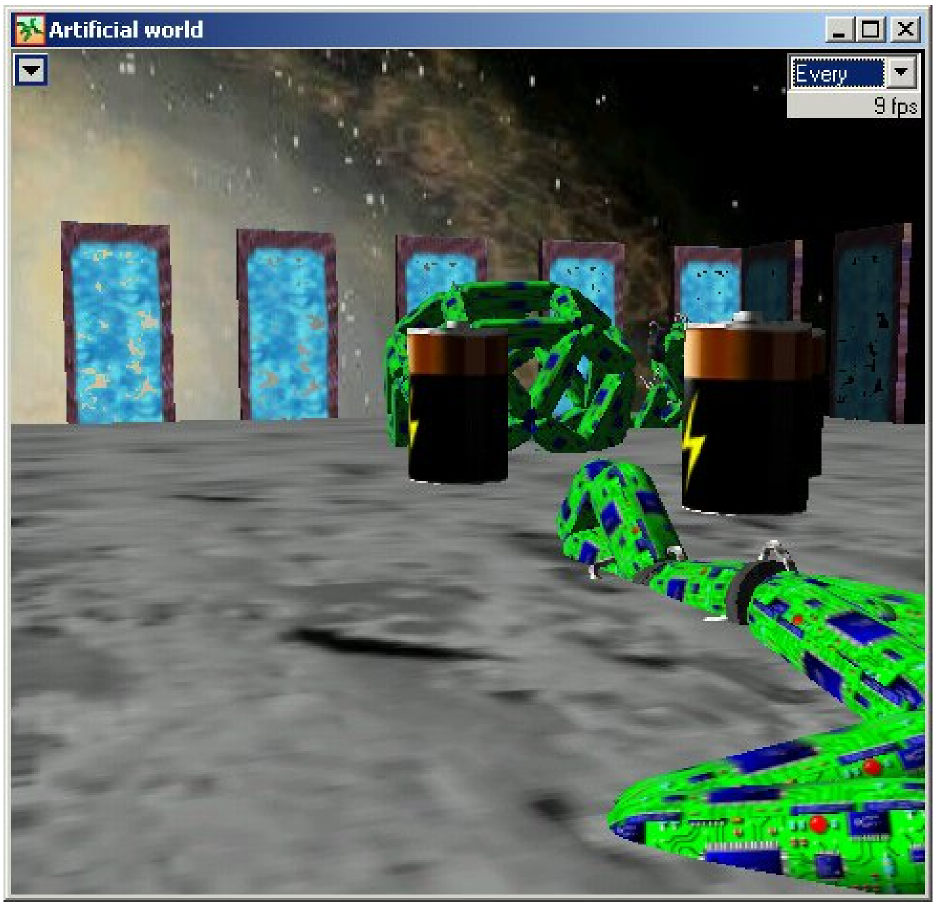
Spooksticks

(description here)



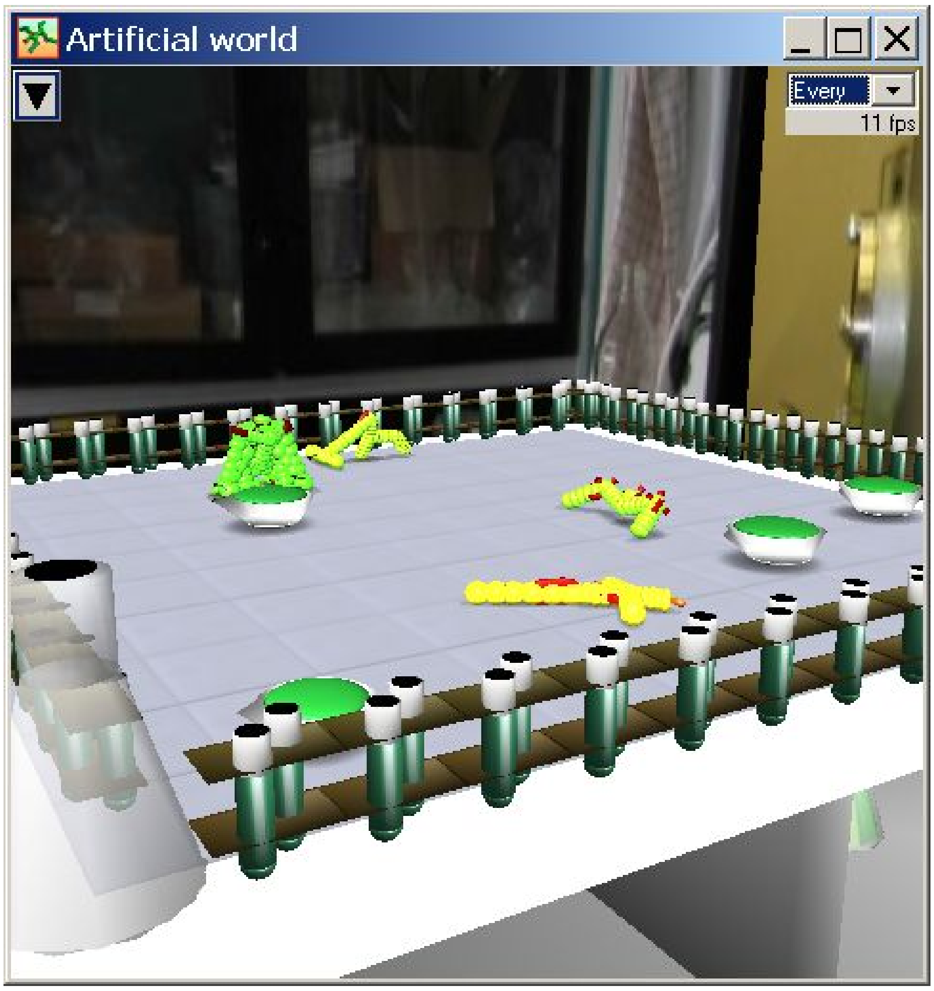
Space

(is there anything more beautiful than infinite space?)



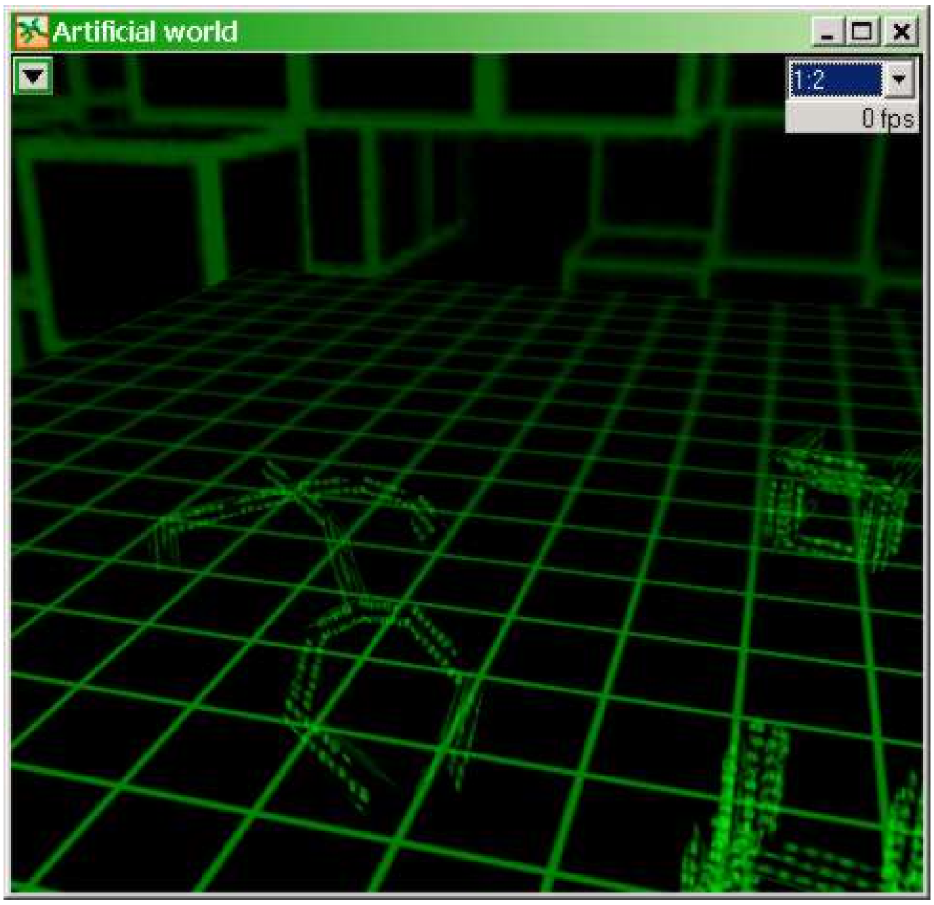
Laboratory

(biochemical laboratory)



Matrix

(no one can be told what the matrix is. You have to see it for yourself...)



16 User Interface – command line

Framsticks command line interface is free. It is faster and simpler than the GUI (no overheads), and it is recommended, especially for long-term experiments.

To learn about command-line parameters, run `frams -?`

16.1 Interactive command-line

To run the interactive command-line program (with its own shell), just execute it. (For Windows console, run `frams.exe`; for UNIX/Linux, run `frams`).

Enter `help` to see help, enter `lm` to see more options. See *frams.ini* file, too. Note that you can access class browser (it is also available in Framsticks GUI).

To quit the program, enter `qu`, or the End-Of-Line character (`^Z` for Windows, `^D` for UNIX/Linux).

16.2 Command-line

You can use arguments, for example:

```
frams "lo e0.gen" "st 100000000" "sa last.gen" qu
```

means that you

- run the program (it always loads default parameters from *default.sim*)
- load *e0.gen* (a file with genotypes)
- simulate 100 000 000 steps
- save genotypes to *last.gen*
- quit (qu)

Quotation marks are needed for parameters which contain spaces. Another example:

```
frams "lo exp_03.expt" "go"
```

means that you

- run the program (it always loads default parameters from *default.sim*)
- load *exp_03.expt* (a file with genotypes and simulation settings)
- run simulation until the Control-C (or Control-Break) is pressed

Thus, if the *exp_03.expt* activates "autosave" option, then you will have periodic snapshot files of the simulation state. You can examine them with the GUI without stopping the command-line simulation, which can be interrupted by pressing control-C at any time.

16.3 Batch

You can use batch mode, for example:

```
frams < commands.txt
```

where *commands.txt* file contains the commands you would enter interactively.

16.4 User-defined actions, scripts and macros

See *cliutils.ini* file for more examples.

17 Script writing

The Framsticks system can interpret commands written in a simple language (FramScript). FramScript can be used for a range of tasks, from custom fitness functions, macros, and user-defined neurons, to user-defined experiment definitions, creatures behaviors, events, and even 3D visualization styles.

Understanding FramScript will let you exploit the full potential of Framsticks, because your scripts can control the Framsticks system. The following documents are related to script writing:

- FramScript syntax
- Contexts, objects, methods, and variables – reference. Temporarily off-line, use Framsticks GUI or CLI.
- Experiment definitions
- Script writing: pdf
- Little how-to: pdf
- FramClipse: for easier scripting!

See also the contents of the "scripts" subdirectory, and the *.ini files in the command-line interface distribution.

18 FramScript language

The FramScript syntax and semantics is very similar to JAVA, C, C++, PHP, etc. In FramScript,

- all variables are untyped and declared using "var" or "global" statements
- functions are defined with the "function" statement
- references can be used for existing objects
- no structures and no pointers can be declared
- there is the Vector object which handles dynamic arrays
- FramScript code can access Framsticks object fields as "Object.field"

Before execution by FVM (Framsticks Virtual Machine), FramScript source code is first translated to FVM assembler source code, and then transformed to FVM machine code. Scripting is a powerful instrument and lets an advanced user control the Framsticks environment and use it according to their needs. However, error reporting while processing scripts is not perfect: if a script is erroneous, do not expect the translator to always precisely locate the bug.

18.1 Statements

(brackets [] denote optional parts)

```
if (expression) statement; [else statement;]
for (expression;expression;expression) statement
while (expression) statement;
do statement; while (expression);
switch(expression) { case constant: statements... default: statements... }
{ statement; [statement;] }
return [expression];
break [level];
continue [level];
label_name:
goto labelname[:];
var name[=expression][,name2[=expression2]][...];
function name([parameter_names]) {statements}
@include "filename"
asm { source code for FVM }
```

Remarks:

- "level" is a positive integer specifying the loop you want to "continue" or "break". "break 1;" is equivalent to "break;", "break 2;" refers to the loop containing the innermost loop, and so on.
- "goto" can be dangerous is some cases (like variable declarations jumped over), so it is recommended to avoid it.

18.2 Variables

Variables must be declared before they are used. Unlike in C, the declaration does not specify the variable type.

Variables are untyped, but the values are typed. The value can be of one of 5 types: integer, floating point, string, object reference or the special empty value (null).

Some operators act differently depending on the value type.

Example:

```
var int=123, float=1.23, string="123";
int+=0.0001;     // result = 123
float+=0.0001;   // result = 1.2301
string+=0.0001;  // result = "1230.0001"
```

That's why sometimes you will need to use constructs like ""+float or 0+string, in order to explicitly force type conversion. Note that the result may be of different type depending on the order of variables (int+string will be integer, float+int will be floating point, int+float will be integer, etc.). Consequently, float+" "+int will produce a single floating point value, whereas ""+float+" "+int will produce a string of two values separated by a space char.

Global variables are declared using the "global" statement. Unlike the local ("var") variables, globals preserve their values between FramScript calls. This is useful when a function is called from outside the code module, as it happens in experiment definitions, styles, neurons and user scripts.

Example:

```
global counter;
function eventHandler()
{ counter=1+counter; }

function getCount()
{ return ""+counter; }
```

18.3 Expressions

Most of the C language expressions are also valid in FramScript. This includes arithmetic and logical operators, parentheses, function calls, and assignments. Moreover, the following FramScript specific opearator are available:

- "typeof" expression – determine the expression value type. The return value should be interpreted as follows:
  - 0 = empty (null)
  - 1 = integer
  - 2 = floating point
  - 3 = string
  - "classname" = object of class "classname". Thus **typeof typeof expression** is equal to **3** if and only if the **expression** returns object reference
- "." (dot) – member access operator (like in C++). It can be applied to any expressions containing object references, or to the special static object names. Static object names are equal to the class names shown in the Class Navigator, e.g.:

  ```
  var expdefname=Simulator.expdef;   // access object field
  Simulator.load("experiment.expt"); // invoke object method
  var two=Math.sqrt(4);              // invoke object function
  ```

  There is only one "static object" for each class. Some objects usually have a single instance and are accessed using this method (like the Simulator or Math in the example above).
  Multiple objects of the same class are accessed indirectly with using object references:

  ```
  var g=LiveLibrary.getGroup(0);            // get the reference
  var info="This group is called "+g.name; // use the reference
  ```

Sometimes, the static object is used to create more objects:

```
var v=Vector.new();           // static object creates a dynamic object
v.add(123);                   // the reference is used
var s=v.size;
```

"ClassName.*" can be used as a reference to the special static object if it has to be passed to some function:

```
var v=callNew(Vector.*);
var d=callNew(Dictionary.*);

function callNew(obj)
{return obj.new();}
```

- ".[expr]" – indirect member access operator. Works like the regular "dot" operator but the member name is passed as an expression:

```
var which=2;
var value=Genotype.["user"+which]; // works like Genotype.user2
```

- `[]` – vector operator (vector is a simple array-like object)
  The [] operator can be used in two ways:
  - ♦ `[ expr1, expr2, ... ]` – vector creation expression. It is a short form of:

    ```
    v=Vector.new(); v.add(expr1); v.add(expr2); ...
    ```
  - ♦ object [expr] – "get" access operator. Works the same as `object.get(expr)`. Thus you can access a Vector object like a regular array in C:

    ```
    var v=["one","two","three"];
    var third=v[2];         // == v.get(2)
    var d=Dictionary.new();
    d.set("name","value");
    var value=d["name"];    // works for any object implementing the "get" method
    ```

  Note that "object [expr] = expression" will not work, because [] acts as "get". You need to use "set" explicitly to alter the vector object.

```
var v=[];                // empty vector
v.set(10,"10th value");  // v.set(...) will expand the vector when required
```

Note 2: See the difference between the "get" operator "[]" and the indirect member access operator ".[]"

```
var d=Dictionary.new();
d.set("size",123);
var v1=d["size"]; // ==123 - equivalent of d.get("size"), gets the previously set value
var v2=d.["size"]; // ==1 - equivalent of d.size, gets the object field value
```

- `function FUNCTIONNAME` – creates the function reference.
- `call (FUNCTIONREF)([arguments,...])` – invokes the function using the function reference obtainad from the "function" operator:

```
var calltable=[function one,function two,function three];
var selector=1;
var fun=calltable[selector];
var result=call(fun)(argument1,argument2); // function two will be called
```

18.4 Functions

Functions are defined as follows:

function name([parameter_names]) {statements}

All parameters are passed by value (copied). However, an argument (or a return value) can contain object reference, in which case it is the reference that is actually copied.

A function can return a value to the caller using the "return expression;" statement. If a bare "return;" is used, the return value is undefined.

Example:

```
function factorial(n)
{
    if (n<3) return n;
    return factorial(n-1)*n;
}
```

Functions can also have multiple names, which is useful when you need many functions with an identical body – for example:

```
function onLeftClick,onMiddleClick,onRightClick(x,y)
{
  //do something...
}
```

18.5 Using Framsticks classes, methods, and variables

See FramScript reference to learn about internal Framsticks classes, methods, and variables. You can also use the class browser (available both in GUI and command-line interfaces).

19 What is an experiment definition, and how to create your own *expdef*

The most important feature of Framsticks is that you may define your own rules for the simulator. There are no predetermined laws, just a script   an *experiment definition*. This script is a set of instructions in some language, which is interpreted by Framsticks program and executed.

This script defines behavior of the Framsticks system in a few associated areas:

- Creation of objects in the world. The script defines where, when and how much of what objects will be created. An object is an evolved organism, food particle, or any other element of the world designed by a researcher. Thus, depending on some specific script, food or obstacles might appear, move and disappear, their location might depend on where creatures are, etc.
- Objects interactions. Object collision/contact is an event, which may cause some action defined by the script author. For example, contact may mean energy ingestion, pushing each other, destruction, or reproduction.
- Evolution. A steady-state (one-at-a-time) selection model, where a single genotype is inserted into a gene pool at a time, can be used. But a standard (i.e. generational replacement) evolutionary algorithm approach is also possible (a new gene pool replaces the whole old gene pool). Another possibility is tournament competition for all pairs of genotypes. In general, the script can define many gene pools and many populations, and perform independent evolutions under different conditions.
- Evaluation criteria are flexible, and do not have to be as simple as the performances supplied by the simulator (but they will have to be based somehow on performances). For example, fitness might depend on time or energy required to fulfill some task, or degree of success (distance from target etc.).

So one can use Framsticks for simulation of various ecosystems, with very diverse laws. This system is very versatile! The script is built from "procedures" assigned to system events. Currently there are the following events:

- onExpDefLoad   occurs after experiment definition was loaded. This procedure should prepare the environment, create gene pools and populations.
- onExpInit   occurs at the beginning of the experiment.
- onExpSave   occurs on save experiment request.
- onExpLoad   occurs on load experiment request. After this event, system state should resemble the state before onExpSave.
- onStep   occurs in each simulation step.
- onBorn   occurs when a new organism is created in the world
- onKill   occurs when a creature is removed from the world
- on[X]Collision   occurs when an object of group [X] has touched some other object.

Thus a researcher may define the behavior of the whole system by implementing appropriate actions within these events. A single script (experiment definition) may use parameters, so it usually allows to perform a whole bunch (class) of diversified experiments.

19.1 Illustrative example (standard experiment definition)

The file "standard.expdef" contains the full source for the script used to optimize creatures on a steady-state basis, with fitness defined as a weighted sum of their performances (see parameters description). This script is quite versatile and complex. Below its general idea is explained, with simplified actions assigned to system events:

`onExpDefLoad()`

- create single gene pool "Genotypes"
- create two populations "Creatures" and "Food"

`onExpInit()`

- empty all gene pools and populations
- place the beginning genotype in "Genotypes"

`onStep()`

- if too little food: create new object in "Food"
- if too few organisms: select a parent from "Genotypes"; mutate, crossover, or copy it. From the resulting genotype create an individual in "Creatures"

`onBorn()`

- move new object into a randomly chosen place in the world
- set starting energy according to object type

`onKill()`

- if "Creatures" object died, save its performance in "Genotypes" (possibly creating a new genotype). If there are too many genotypes in "Genotypes", remove one.

`onFoodCollision()`

- send energy portion from "Food" object to "Creature" object.

When you look at the standard.expdef file (in the "scripts" subdirectory), you will see it is written in a simple language. Thus, when creating your own script, computer science background is helpful. A documentation on script writing is available here. The existing scripts should serve as examples (see *cliutils.ini* file in command-line frams distribution for illustrative examples). Any file with the ".expdef" extension, which is present in the "scripts" directory, will automatically appear on the list of experiment definitions to choose from.

20 Programming: contexts and classes summary

FULL INFORMATION IS AVAILABLE IN THE HTML VERSION OF THE CLASSES DOCUMENTATION. THIS IS A SUMMARY ONLY.

The most recent version is always available online. You can download it zip'ped for off-line use.

Framsticks classes are grouped in *contexts*. Context hierarchy is presented as nested boxes in the frame on the left. From a specific context, you can access its objects as well as objects of its parent contexts. Most classes have corresponding static objects that can be accessed as `ClassName.member` (like *static* in C and Java).

Introduction

Index: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

20.1 All contexts and classes

Global context

CheckpointEvent Collision CrCollision Creature CreatureSettings CreatureSignals CreatureSnapshot Dictionary EnergyParticles File FunctionReference GenePool GenePools GenMan GenManStats Geno Genotype Interface Joint Loader Math MechJoint MechPart Model ModelGeometry ModelSimilarity ModelSymmetry Neuro NeuroClass NeuroClassLibrary NeuroDef NeuronsSimEnabled NeuroSignals ODE Orient Part Population Populations POVExport Ref Signal SignalView sim_params Simulator SlaveSimulators stats StopEvent String UserScripts Vector WireframeAppearance World WorldMap WorldSignals XYZ

Experiment definition

ExpProperties ExpState

Neuron definitions

n Neuro NeuroProperties

Fitness formula

this

Visual style definition

Creature Element GeomBuilder GL j Loader m Material MechJoint MechPart n p Scene VertexBuilder VisProperties VisualModel VisualStyle World WorldMap

Signal label formula

Signal

Command line interface

ClassBrowser CLI RowFormat

Framsticks Theater

CreatureCharts CreatureSelection DynaChart DynaChart2D GLDisplay GUI GUIElement ImageExport Material Matrix NeuroDiagram Obj3D Rectangle Shader Sharing ShowManager ShowProperties SimStatsCharts TrackingCam VisualStyle Window

Generated from Framsticks source code, version 5.0rc13 (Fri Mar 30 01:42:33 2018)

20.2 Global context

Objects defined in the global context are available in all other places.

Classes defined in Global context:

- CheckpointEvent
  Used in onSlaveCheckpoint() which is called when a Slave Simulator checkpoint is reached.
- Collision
  Used in collision handlers (On___Collision).
- CrCollision
  Used in creature collision handlers (On___CrCollision).
- Creature
  The object inside the simulated world, including its physical structure, neural network and performance data.
- CreatureSettings
  Creature building parameters
- CreatureSignals
  Signal sources associated with a creature.
- CreatureSnapshot
  A data object consisting of the same fields as the serialized Creature.
- Dictionary
  Dictionary associates stored values with string keys ("key" is the first argument in get/set/remove functions).
- EnergyParticles
- File
  Provides read/write access to the filesystem.
- FunctionReference
  Function reference objects are created using the 'function' operator.
- GenePool
  GenePool objects are accessed by GenePools[index], or Genotype.
- GenePools
  Manages all genotypes in the experiment, organized in one or more groups.
- GenMan
  Manages various genetic operations, using appropriate operators for the argument genotype format.
- GenManStats
  Statistics for genetic operations.
- Geno
  All information about a single genotype.
- Genotype

A Genotype with the associated performance information.
- Interface
  Used to query for object member descriptions.
- Joint
- Loader
  Support for loading files in the Framsticks format.
- Math
  Mathematical functions library.
- MechJoint
- MechPart
- Model
- ModelGeometry
  Approximately estimates sizes, volume, and area of a Model based on the geometry of its parts.
- ModelSimilarity
  Evaluates morphological dissimilarity.
- ModelSymmetry
  Calculates bilateral symmetry.
- Neuro
  Live Neuron object.
- NeuroClass
  The static NeuroClass object refers to the class selected in the NeuroClassLibrary.
- NeuroClassLibrary
  Set of Neuron classes.
- NeuroDef
- NeuronsSimEnabled
- NeuroSignals
  Signals attached to a neuron.
- ODE
  ODE Parameters.
- Orient
  3D orientation, stored as 3x3 matrix.
- Part
- Population
  A set of Creature objects, sharing some high level simulation properties (performance calculation, NN simulation, collision detection, event handling).
- Populations
  Manages all Creature objects in the experiment, organized in one or more groups.
- POVExport
- Ref
  Reference objects.
- Signal
  Signals broadcast information in a channel (being an abstract communication medium that could be imagined as sound, smell, vision or anything else).
- SignalView
  This object can be used by an Experiment Definition or a Framsticks Theater show script to configure the preferred Signal visualization in the experiment.
- sim_params
  This object groups all simulator parameters so they can be loaded or saved with a single call (see scripts/standard_loadsave.
- Simulator
  The Framsticks simulator.
- SlaveSimulators

This is a vector of slave Simulator objects.

- stats
- StopEvent
  Used in onSlaveStop() which is called when a Slave Simulator is stopped.
- String
  String functions library.
- UserScripts
- Vector
  Vector is a 1-dimensional array indexed by an integer value (starting from 0).
- WireframeAppearance
  This object defines appearance of the elements of creatures (applies to the 'wireframe' display mode).
- World
  Environment properties.
- WorldMap
  Environment details for "Blocks" and "Heightfield" world type.
- WorldSignals
  Use this object to create stationary signals (not associated with any moving object) and to receive signals from any location in the world.
- XYZ
  3D vector

Global context

Experiment definition Neuron definitions Fitness formula Visual style definition Signal label formula Command line interface Framsticks Theater

20.3 Experiment definition

ExpProperties and ExpState objects are created according to "property:" and "state:" sections in the expdef file.

Classes defined in Experiment definition:

- ExpProperties
- ExpState

Experiment definition

Global context >> Neuron definitions Fitness formula Visual style definition Signal label formula Command line interface Framsticks Theater

20.4 Neuron definitions

Neuron definition consists of 3 parts:
- custom fields definition ("property:" sections, available as "NeuroProperties" object)
- "function init()" (will be called at the neuron creation time)
- "function go()" (should set new Neuro.state value)

Classes defined in Neuron definitions:

- n
- Neuro
  Live Neuron object.
- NeuroProperties

Each neuron contains a reference to the "NeuroProperties" object with neuron's own custom fields (also known as parameters or properties).

Global context >> Experiment definition >> **Neuron definitions**

20.5 Fitness formula

Fitness formula calculates genotype fitness. You can access Genotype fields using "this" object.
Example: return this.velocity*100;

Classes defined in Fitness formula:

- this
  A Genotype with the associated performance information.

Global context >> Experiment definition >> **Fitness formula**

20.6 Visual style definition

The style definition comes from the "style" files. It defines the appearance of the OpenGL display. VisParams properties are created according to the "property:" sections in the file.

Framsticks calls user defined functions to create and update the object's visual elements during the simulation.

Each function name is built as follows: "[<element_Vstyle>_]<model_Vstyle>_<element>_<action>"

- action="build" when the visual model is first created (load or create geometry and set initial attributes)
- action="update" each time the visual model is displayed (modify geometry to reflect current position or attributes)

"<element>" refers to the kind of the element being processed (part, joint, neuron, model).
The script can access different objects depending on the kind:
1. part
can access:
- "MechPart" after calling Element.useMechPart()
- "Creature" after calling Element.useCreature()

2. joint
can access:
- "MechJoint" after calling Element.useMechJoint()
- "MechPart" after calling Element.useMechPart1() or Element.useMechPart2()
- "Creature" after calling Element.useCreature()

3. neuro
can access:
- "n" after calling Element.useNeuro()
- "Creature" after calling Element.useCreature()

4. model
can access:
- "Creature" after calling Element.useCreature()

"<element_Vstyle>" and "<model_Vstyle>" are element's and model's Vstyle fields. <element_Vstyle> is omitted when the field is empty. Empty model's Vstyle is replaced with "default". Both <element_Vstyle> and <model_Vstyle> can contain several words separated by "_". This is interpreted as a kind of subclassing, i.e. Framsticks will automatically look for the general version of the function when the specific one is not found.
Example:
Assuming a part with Vstyle="green" in a model with Vstyle="food" the following functions will be tried until a match is found:
- green_food_part_...
- food_part_...
- default_part_...
So if the experiment requests visualizing some element as "green food" but the visual style only defines generic form of "food", Framsticks will automatically choose the best available visualization.

Classes defined in Visual style definition:

- Creature
  The object inside the simulated world, including its physical structure, neural network and performance data.
- Element
  Information about current visual element: Part, Joint or Neuro.
- GeomBuilder
  Scene graph access (build and manipulate the 3D object tree).
- GL
  OpenGL constants used in GeomBuilder and Material functions.
- j
- Loader
  Loads 3d objects files.
- m
- Material
  Manipulate 3d objects surface properties (Material objects can be associated with geometry nodes).
- MechJoint
  Current joint properties during the simulation
- MechPart
  Current part properties during the simulation
- n
- p
- Scene
  Additional properties to be used in the current view
- VertexBuilder
- VisProperties
- VisualModel
  Visual representation of the Creature object.
- VisualStyle
- World
  Environment properties.
- WorldMap
  Environment details for "Blocks" and "Heightfield" world type.

Global context >> Experiment definition >> **Visual style definition**

20.7 Command line interface

Command line objects are available in "frams" and "theater".

Classes defined in Command line interface:

- ClassBrowser
  Object reference, can be used to provide online hints.
- CLI
  Commandline support functions.
- RowFormat
  Used for creating simple listings of objects (in command line applications).

Global context >> Experiment definition >> **Command line interface**
Framsticks Theater

21 Software package remarks

The what's new list is available.

21.1 Files and file types

All of the Framsticks data files are text files, so you can view and edit them with any text viewer/editor (notepad, wordpad, type, more, vi, pico). However, manual modification of most of these files is not recommended and should not be needed. The following files are used:

Type	Contents	Location
GEN	genotypes	any
SIM	settings for the given experiment definition	any
EXPT	combination of SIM and GEN contents (a "snapshot" of some experimentation moment)	any
EXPDEF	scripts which define experiment definitions	"scripts" subdirectory
NEURO	scripts which define custom neurons	"scripts" subdirectory
SCRIPT	any user-defined scripts	"scripts" subdirectory
STYLE	scripts which define visual styles for OpenGL graphics	"3dobj" subdirectory

The detailed technical documentation of file and data formats used in Framsticks is available here.

21.2 Sample files provided

- 'default.sim' keeps default ("factory") parameters for the simulator. The file is always loaded when the program starts.
- 'f-x.sim' describes more sophisticated world and more complex simulation rules (not intended for evolution – just to play with Framsticks and enjoy simulation).
- 'manual.sim' is good for testing (creatures do not die etc.).
- 'speed.sim' is good for velocity-oriented evolution on the ground. It is originally identical with the 'default.sim'.
- 'swim.sim' is good for velocity-oriented evolution in water.
- 'demo-chase.sim' is a setup where creatures try to chase each other and eat food. Use with 'demo-chase.gen'.

21.3 Release notes

21.3.1 Simulator

- 'W' and 'w' (weight) modifiers are temporarily disabled (they work under water only)
- in v2.x: destructive collisions, missing from v1.78

21.3.2 MS Windows graphical user interface

- Windows 98/ME or 2000/XP recommended. For Windows 95/98/NT you may need to install upgrades/service packs to fix bugs in Windows OpenGL
- Windows NT does not display icons correctly

21.3.3 Amiga graphical user interface (old)

- camera/World Display settings are not saved
- problems with window refreshing
- the program was not tested with Kickstart 2.0, which may cause some problems
- OpenGL window is for viewing only (camera can be moved in World window)

21.4 Reporting bugs

Please report any bugs to support@framsticks.com. When reporting bugs, be sure to mention these things:

- the name and version of the Framsticks program which causes problems,
- operating system you are running (with/without service pack, which?),
- if the bug is repeatable, and under what conditions? Try to find the simplest sequence of steps to reproduce the problem,
- parameters of the simulator when the error occurs,
- attach files pertinent to reproducing the bug, like settings, genotypes, and your own scripts,
- enclose the "messages.out" file, if some errors are listed there.

22 Events and FAQ

22.1 Framsticks-related events

- April 21, 1998: Demonstration on a DEMO session of European machine learning conference, Chemnitz, Germany; a paper.
- November 1998: Netherlandish CD-Magazine, C'TROM 18 – demo movies and pictures, project description.
- November 17, 1998: CALResCo awards this site as one of the best visual Artificial Life resources available on the web.
- March 1999: Chip and Chip CD Polish edition (3/99) – an article, WWW site, application.
- March 17, 1999: an interview for Radio BIS (Poland) – artificial intellgence and artificial life.
- March 1999: an interview for Reporter (Poland) – artificial life and artificial intellgence.
- May 25-28, 1999: Polish conference "Evolutionary Algorithms and Global Optimization" (a paper).
- September 13-17, 1999: Poster, demonstration and paper in the 5th European Conference on Artificial Life, Lausanne, Switzerland. M. Komosinski awarded for the outstanding work in the field of Artificial Life.
- September 1999: Interview for L'Hebdo, N° 38/99 (Internet version of the article – in French – available here).
- October 15, 1999: An open lecture/presentation/demo on "Evolutionary Algorithms and Simulations of Life", Poznan Univ. Tech., Poland.
- November 19, 1999: Framsticks presentation at the LEGO Lab, Aarhus, Denmark.
- January 7 and 14, 2000: Interviews/Articles in a Polish newspaper, Glos Wielkopolski.
- February 2000: Interview for RMF FM radio.
- April 11-14, 2000: INFOSYSTEM 2000 presentation. Country-distributed evolution, remote ONYX2 graphical workstation visualization (OpenGL), parallel rendering.
- June-July 2000: Evolutionary Art Exhibition "Darwin's coach", Geneva. Framsticks presentation.
- July 2000: Framsticks Experimentation Center prototype.
- July 2000: Presentation at the 2nd International Conference on Virtual Worlds, Paris, France. A paper.
- August 2000: Presentation at the 7th International Conference on Artificial Life, Portland, USA. A paper.
- October 11-14, 2000: Presentation at the sgi2000: SGI Users' Conference, Krakow, Poland. Country-distributed evolution, remote ONYX2 graphical workstation visualization (OpenGL).
- December 1-2, 2000: TheoLab – Research Unit for Structure Dynamics and the Evolution of Systems, Jena, Germany. Presentation on the workshop "Evolution and Neural Control of Autonomous Systems".
- December 4, 2000: Seminar at the Chair of Systems Analysis, Department of Computer Science, University of Dortmund, Germany.
- May 14-18, 2001: Seminar at the meeting "Complexity-Unifying Themes for the Sciences and New Frontiers for Mathematics" of Santa Fe Institute for the Sciences of Complexity (SFI) and Max Planck Institute for Mathematics in the Sciences (MPI). Leipzig, Germany.
- September 10-14, 2001: Presentation and paper on the 6th European Conference on Artificial Life, Prague, Czech Republic.
- November 13/14, 2001: Framsticks in the Polish TV station, TVN. See snapshots.
- December 2001: Two papers: for *Artificial Life Journal* and for *Theory in Biosciences*.
- February 2002: Article in *Software 2.0* (a computer magazine) about Alife, AI, and Framsticks.
- April/May 2002: Articles/news in internet information services: PAP, Chip, WP, ONET. Interviews for radio: III P.R., TOK FM, Radio Krakow. Television: WTK.
- March 2003: "Data Ecologies" workshop in Austria, "the Framsticks day".
- September 2003: an interview for the Chip magazine.

- September 2003: International Conference on Computational Intelligence and Natural Computing, North Carolina, USA. A paper and presentation.
- March 2004: An article for the "Scientific American" Polish edition.
- July 2004: European Framsticks summer school in Lodz, Poland.
- February 2005: Framsticks and the *Framsticks deathmatch* experiment described in the "Software 2.0 Extra" magazine.
- June 2005: Framsticks chapter in the Artificial Life Models in Software book.
- November 2005: Vector eye and sensory-motor coordination experiments presented on the cognitive science conference.
- August-October 2006: Invited lectures at the University of North Carolina at Charlotte, USA.
- September 2006: Interview for biota.org group.
- November 2006: A paper introducing biologically-inspired sensory-motor coordination model, KES 2006: International Conf. on Knowledge-Based & Intelligent Information and Engineering Systems, Bournemouth, UK.
- December 2006: Framsticks workshop in Poznan, Poland.
- April 2007: Invited lecture at the Robots Festival, Poznan, Poland.

22.2 Frequently Asked Questions

- Who can benefit from using the program/Why is it useful for me?
- The program is a very good educational tool. As most of the information needed to use the program is available on these pages, the software will let you experiment with complex stick structures ("alive" or not), complex neural networks and a three-dimensional, water and land simulation. The application has a special interface that you will quickly learn and use to build structures, test them, evolve, and improve. If you study computer science, biology, robotics, physics, mechanics, etc., the program may be a nice way of learning various topics. If you are a scientist, you may want to explore the capabilities of the software and use them in a more advanced way (custom scripts and experiments). If you are interested in computer graphics, you can make wonderful movies and pictures. You can also use our program to compete/play with other users (for example, who will grow or design a faster or stronger creature?) and for entertainment.
  You can also take part in a discussion to seek help or express your ideas.
- How can I contribute?
- You can conduct lots of experiments on your own, from very simple (speed-oriented) to very sophisticated with unpredictable results (spontaneous/open-ended evolutions). You can send us your creatures, and you can also take part in the challenges (see here, you may be rewarded!). You can also design your own 3D visualization styles, create your own experiments, and help us with translation of the web pages. Or develop helper Framsticks software. If you are interested, please contact us.
- Does this project have any practical applications?
- The project in its current form is not used in industry. However, every technique used in Framsticks has lots of applications: evolutionary algorithms, optimization methods, neural networks, finite element method and the mechanic simulation etc. This research can be useful for people working in computer science, robotics, biology, physics and many other scientists, and for non-professionals too. The program can be useful for all who are interested in any of the techniques used, also within a "to learn by playing" approach.
  The most important goal is to use this program to perform various evolutionary experiments in a complex environment. The study of achieved results helps understand better evolutionary mechanisms and yields interesting conclusions concerning behavior and structure of emerging creatures.
- Why is the program in English only?
- Web site translation is very time-consuming. Synchronization of continuous (and numerous!) corrections is a never-ending work. English is the most common scientific language, that is why it was chosen.
- How long does one evolution take?

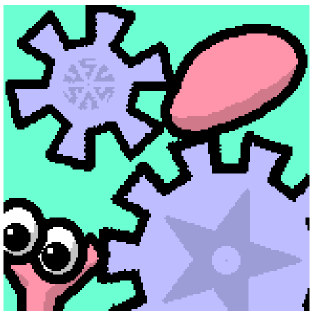
- A speed-oriented evolution without speciation is relatively short. Assuming that the process begins with no additional knowledge (starts with a single, simplest organism genotype), the methods of movement are usually discovered after several minutes. After a few hours, creatures are quite efficient in locomotion. More sophisticated evolutions (e.g. spontaneous/open-ended) take longer. Of course, it is crucial to adjust parameters properly in order to efficiently use the simulation time.
- I would like to add a link to your project from my web pages.
- Use http://www.framsticks.com/
  You can add a picture:

http://www.framsticks.com/files/common/artlife-evol.gif

http://www.framsticks.com/files/common/artlife-mini.gif

http://www.framsticks.com/files/common/artlife-midi.gif

http://www.framsticks.com/files/common/artlife.gif


http://www.framsticks.com/files/common/artlife-maxi.gif

If you need very high-quality graphical materials concerning this project (for printing, video, etc.), do contact us.

23 Framsticks resources (including survey results)

If you have some kind of material to be included here, let us know!

23.1 Documentation

- Framsticks Manual: PDF file (includes the tutorial)
- Framsticks Tutorial, step by step
- A graphical interface (GUI) mini-tutorial
- A short tutorial about Framsticks GUI: PDF file
- Scripts in Framsticks: useful resources.

23.2 Exercises and experiments – for teachers and students

- Sample student exercises for the Bio-inspired Adaptive Machines course, from Swiss Federal Institute of Technology, Lausanne (EPFL) - see also bottom of that page for a specification of a few exercises more
- Interesting experimental exercises from the Virtual Life Lab, including the well-documented deathmatch educational challenge
- A number of various tasks and ideas

23.3 Selected publications and their BibTeX entries

- M. Komosinski, Sz. Ulatowski, Framsticks: sztuczne zycie – zlozona symulacja stworzen i ich ewolucji. W: *Materialy konferencyjne III Krajowej Konferencji Algorytmy Ewolucyjne i Optymalizacja Globalna KAEiOG (Proceedings of Evolutionary Algorithms and Global Optimization Conference)*, Potok Zloty, Poland, May 25-28, 1999, 157-166. [In Polish]
- M. Komosinski, Sz. Ulatowski, Framsticks: towards a simulation of a nature-like world, creatures and evolution. In: *Proceedings of 5th European Conference on Artificial Life (ECAL99)*, September 13-17, 1999, Lausanne, Switzerland, Springer-Verlag (LNAI 1674), 261-265.
- M. Komosinski, The World of Framsticks: Simulation, Evolution, Interaction. In: *Proceedings of 2nd International Conference on Virtual Worlds (VW2000)*, Paris, France, July 2000. Springer-Verlag (LNAI 1834), 214-224.
- M. Komosinski, A. Rotaru-Varga, From Directed to Open-Ended Evolution in a Complex Simulation Model. In: *Proceedings of 7th International Conference on Artificial Life (ALIFE7)*, Portland, USA, 2000. MIT Press, 293-299.
- A. Adamatzky, Software review: Framsticks, *Kybernetes: The International Journal of Systems & Cybernetics*, **29** (9/10), 2000. MCB University Press, 1344-1351.
- M. Komosinski, M. Kubiak, Taxonomy in Alife. Measures of Similarity for Complex Artificial Organisms. In: *Proceedings of 6th European Conference on Artificial Life (ECAL01)*, September 10-14, 2001, Prague, Czech Republic, Springer-Verlag (LNAI 2159), 685-694.
- M. Komosinski, G. Koczyk, M. Kubiak, On estimating similarity of artificial and real organisms. In: *Theory in Biosciences*, **120**, December 2001, 271-286.
- M. Komosinski, A. Rotaru-Varga, Comparison of Different Genotype Encodings for Simulated 3D Agents. In: *Artificial Life Journal*, **7** (4), 2001. MIT Press, 395-418.
- P. Mandik, Synthetic Neuroethology. In: *Metaphilosophy*, **33** (1/2), January 2002. 11-29.
- P. Mandik, Varieties of Representation in Evolved and Embodied Neural Networks. William Paterson University Cognitive Science Technical Report 2002-03.
- M. Komosinski, The Framsticks system: versatile simulator of 3D agents and their evolution. In: *Kybernetes: The International Journal of Systems & Cybernetics*, **32** (1/2), 2003. MCB University

Press, 156-173.
- M. Hapke, M. Komosinski, D. Waclawski. Application of evolutionarily optimized fuzzy controllers for virtual robots. In: *Proceedings of the 7th Joint Conference on Information Sciences (JCIS7)*, September 2003, North Carolina, USA, Association for Intelligent Machinery, 1605-1608.
- M. Komosinski, Sz. Ulatowski, Genetic mappings in artificial genomes. In: *Theory in Biosciences*, 2004.
-

A. Adamatzky, M. Komosinski (Eds), Artificial Life Models in Software. Springer-Verlag, 2005. ISBN 1-85233-945-4.
- W. de Back, M. Wiering, E. de Jong. Red Queen dynamics in a predator-prey ecosystem. In: *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 2006, Seattle, Washington, USA. ACM Press, 381-382.
- W. de Back. Master's thesis, Eco-evolutionary experiments with situated agents, 2006.
- M. Hoffmann. Master's thesis, Structural coupling with environment and its modelling on neural driven agents, 2006.

23.4 Related web pages and projects

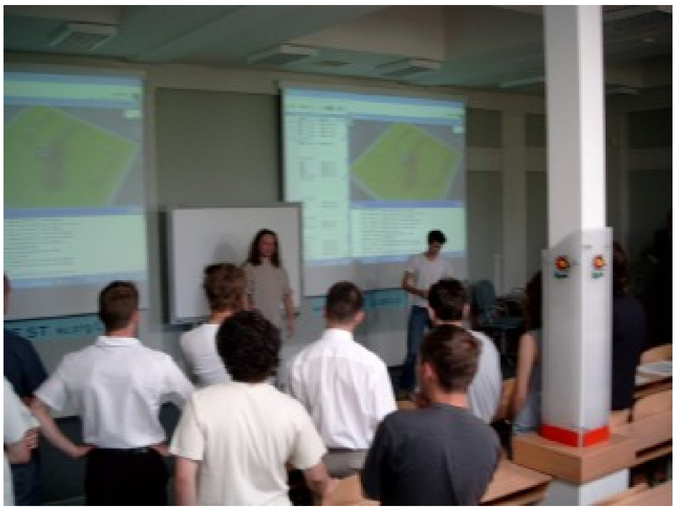
- Framsticks Experimentation Center (experiments and genotypes database, news forums and more).
- Wiki (projects, news).
- Virtual Life Lab in Department of Philosophy & Robotics Lab, Utrecht University. See also Experiments in Synthetic Evolutionary Psychology.
- From povray sequence to blender: Importing animation of creatures / Importer l'animation des créatures de framstick
- Links to artificial life sites: see the "Links" section of the artificial life portal, alife.pl.
- [In Polish] Strona Polskich Uzytkownikow Framsticks.

23.5 Meet some Framsticks users

A lab class. Cognitive Science Laboratory, Department of Philosophy, William Paterson University of New Jersey/USA.

The symposium. Department of Philosophy & Robotics Lab, Utrecht University/The Netherlands.

Lab classes. European Summer School, Lodz/Poland.


The final competition. European Summer School, Lodz/Poland.

Adaptive Machines course, Lausanne/Switzerland.

Adaptive Machines course, Lausanne/Switzerland.

Framsticks workshop, Poznan/Poland.

Framsticks workshop, Poznan/Poland.

Send us a photo of your group!

24 Framsticks Tutorial – step by step

This tutorial is also available in Polish.

About this document

The Framsticks Tutorial provides a step-by-step introduction to Framsticks. It covers basic issues related to program interface, simulation, genetics, and evolution. The tutorial also refers to a few more advanced functions, including writing user procedures, neuron types, etc. This tutorial can be used during Framsticks classes, and for self-education as well. It requires the Framsticks GUI (available to download from the Framsticks web site) installed on an MS Windows operating system.

If you are a teacher, take some time to go through all steps and do the exercises. We recommend reading the Framsticks Manual first. You may want to skip some parts of the tutorial during the class, depending on your discipline. For example, you may focus either on simulation, genetics, evolution, interactions, or script programming. Computer science, robotics, biology, cognitive science or philosophy require emphasis on different sections. It is a good idea to first estimate the time needed for students to perform the tasks. You might also enumerate the goals that students should be able to learn (or be able to test) after they have completed this tutorial. You may ask students to prepare written report (or oral presentation) based on selected exercises.

If you are a student or want to go through this tutorial to learn Framsticks, we recommend that you first browse the Framsticks Manual. Then start with the tutorial. It is a good idea to ask somebody else (a friend) to independently do these exercises as well, and later discuss them together and exchange ideas. If some parts are too difficult, you may skip them, and later go through the tutorial again. Refer to the Manual if you miss some information.

The full documentation is available on the official web site and in the Framsticks Manual. This tutorial is not a substitute for the documentation.

I. Basics

1. Meet Framsticks

   Goal: see a few Framsticks creatures, their behavior, physical properties, interactions, etc.

   Setting: use the Framsticks GUI for Windows. Set the Simulator Parameters/Experiment/Populations/Creatures/Death to "off", enable OpenGL (if available). This presentation can also be performed using the "Framsticks Theater" and the "presentation" show.

   ◊ Load "walking.gen" and start the simulation
   ◊ Simulate some interesting creatures: quadruped, lizard, hopping spider, etc. *(right click on the genotype -> Simulate -> Creatures)*
   ◊ Basic user interface tasks:
   - · left button mouse drag: rotate camera
   - · right button mouse drag: pan
   - · left double click: zoom on creature
   - · wheel up/down: zoom
   - · control + left click: grab creature ("manipulator")

· shift + right click: select action
· right double click: feed (perform first action)

2. Framsticks GUI

Goal: Get to know the Framsticks GUI and basic concepts of the Framsticks simulator

◊ Simulator Parameters: some parameters should be modified while playing with the creatures:
  · `Experiment/Populations/Creatures/Death` should be off. Otherwise the creatures could die unexpectedly.
  · `Experiment/Parameters/Simulated Creatures` the simulator will spawn that many creatures automatically. Set this to 0 if you don't want the simulator to mess with your creatures
◊ Genotype: description (building plan) of the creature. Can be saved in a file. *(GUI: menu -> File -> Load/Save genotypes)*
◊ Body&Brain Window: click on the genotype to see the physical structure and neuron connections
◊ Creature: can be built from genotype
(GUI: right click on the genotype -> Simulate -> Creatures)
(GUI: right click on the creature -> Simulate -> Kill/Delete)
◊ Mechanical simulation: can be MechaStick or ODE (rigid bodies). See the `World/Simulation engine` setting. Try both engines, use the manipulator to play with creature bodies and note differences.

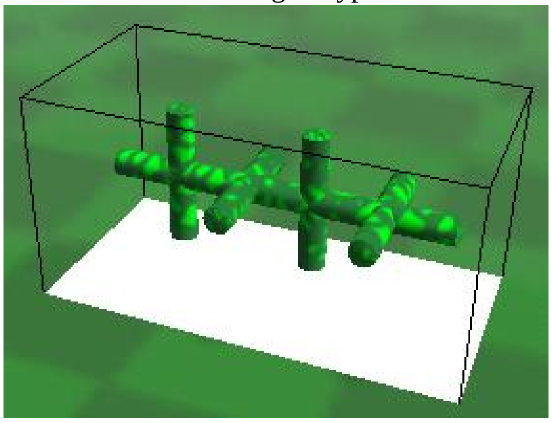
3. Learn genetics

Goal: Learn how to design a simple structure in the 'f1' encoding

◊ Click **New** to create a new genotype
◊ Enter the genotype and watch the results in the structure preview window
(See also the f1 genotype specification in the Framsticks Manual or in the help)
  · Enter **X** a single line segment (stick) appears
  · Enter a few more **X**'s and move the cursor along the genotype. The structure view will highlight the corresponding body element. Click on the sticks to have the corresponding genotype part underlined.
  · Branching: use parentheses **(...)** and commas to create tree-like structures. Example: **XXX(XX,X)**. See how the additional commas influence the branching direction: **XXX(,,,XX,X)**
  · Parentheses can be nested, try: **XXX(XX,X(X,X))** *(go through the genotype with the cursor to make sure you know which stick is created from each X)*
  · Rotate the branching plane: introduce the R/r modifiers. Insert **R**'s one by one and watch the resulting structure: **XXX(XX,RRRX(X,X))**
  · More shape modifiers: Q C L insert into the genotype and guess how they work. See the Framsticks Manual for explanation. Note the general rule: lowercase/uppercase letters have the opposite meaning.
  · Note that red background means that the genotype is invalid like "X[", and pink background means a genotype error which can be corrected automatically like "X()". Some genotypes are valid in syntax, but problems arise during building creatures. Try for example the "X[|][|]" genotype, which has two identical muscles in the same place in body. As this is invalid, an additional entry will appear ("Build problems encountered!") in the small arrow menu in both panels displaying body and brain. Click it to see the list of build problems.

◊ **Exercise 1:** Build an interesting shape using this genotype language.

◊ **Exercise 2:** Guess the genotype from the creature's shape

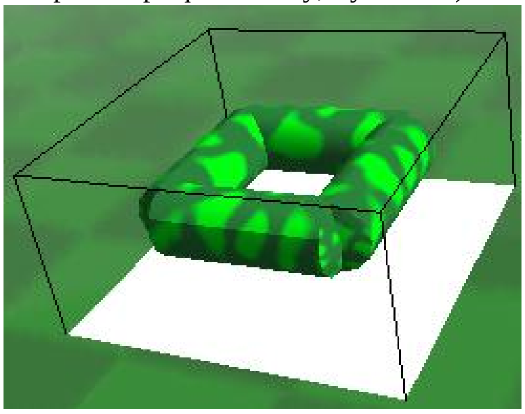


Solution: ask the tutor.

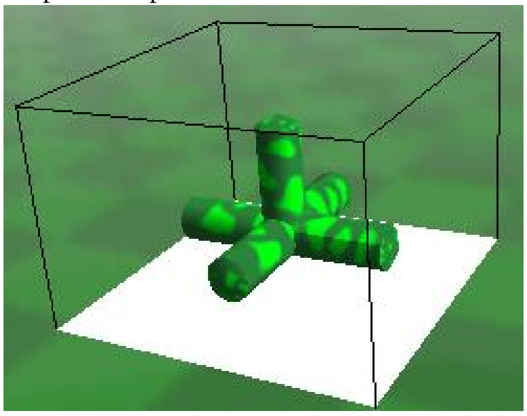
Exercise 3: Can you write another genotype resulting in the same shape?

◊ Conclusion: Limitations of this 'f1' genotype encoding

- Only tree-like structures: **closed** loops not possible (although you can build an open-loop square easily, try to do it).



- The stem and its branches are always placed on the same plane, the following shape is not possible:



(However, those shapes can be built in Framsticks using another genotype encoding, called 'f0')

4. Neurons

Goal: add neurons (effectors, receptors) to the creature

◊ Neural net description is mixed with the body genotype, like this: X[1:2]X[-1:3,0:1]X Each entity enclosed in square brackets is a single neuron. The number before ":" is the neuron reference (0=self connection, +1=next neuron, -1=previous one). Another number (after ":") is the connection weight. No neuron type name is specified, thus the default "N" neuron is assumed. See the following examples and ensure you understand them well. Enter them as genotypes into the program:

· **cccXXXXX[0:0][-1:1][-1:1][-1:1][-1:1]** *(Chain of neurons)*

· **cccX[0:0]X[-1:1]X[-1:1]X[-1:1]X[-1:1]** *(Different neuron placement in the body)*

· **cccX[4:1]X[-1:1]X[-1:1]X[-1:1]X[-1:1]** *(Close the loop)*

· **cccXXXXX[0:0][0:0]X[0:0]X[0:0]X[-1:1,-2:1,-3:1,-4:1]** *(Multiple inputs)*

◊ There are many different neuron types, see "Simulation Parameters: Genetics: Neurons to add" for a quick summary. Examples:

· **X[Sin,f0:0.1]**

"Sin" is a neuron type (Sinus generator)

then you have to write a comma (another, obsolete   and more complex notation does not require this comma, but let's use the simpler and newer notation)

"f0" is the property of the Sin neuron   a base frequency for the sinus generator

· **X[T][G][S][-3:1,-2:1,-1:1]**   *3 receptors connected to a single neuron*

· **XX[T][-1:1][|,-1:1]**   *receptor (touch sensor) -> neuron -> effector (muscle)*

◊ You can have a look at the f1 encoding reference.

5. Brain control

Goal: Monitor and control "living" neurons

◊ Start with a simple version of our previous genotype: **X(X,RRLX(X,X,X),X)** *(one segment instead of three)*

◊ Add two strong muscle neurons. Set the muscle power ("p" property) to maximum (1):

X(X,RRLX[|,p:1][@,p:1](X,X,X),X)

◊ Make sure the simulator parameters are as follows:

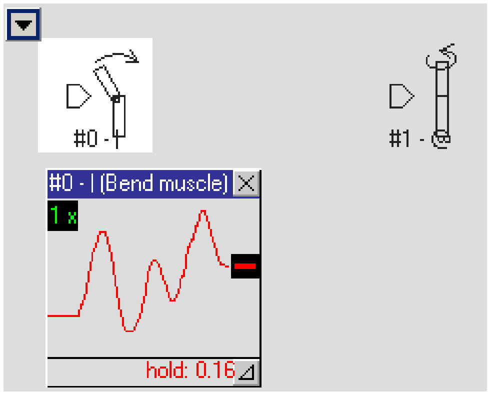
· Experiment/Populations/Creatures: Death is disabled

· Experiment/Populations/Creatures: Neural net simulation is "Immediate"

· Experiment/Populations/Creatures: Performance calculation is "Immediate"

· Experiment/Parameters: Simulated creatures is set to 0

◊ Build a creature from this genotype and start the simulation (Simulate -> Creatures, Simulation -> Run). The creature appears in the world and on the populations list.



◊ Select the creature by clicking on the populations list. This will associate the Body&Brain window with the living creature (and not with the genotype) and zoom the camera on it.

◊ Double-click on the first neuron in the Body&Brain window (on the neural net diagram).

◊ The tiny window inside the diagram is a probe. It can be used to display and change the signal. Drag the thick black handle up and down. You can see the immediate muscle response   the creature moves. Try to find the best signal pattern to make the creature go forward.

◊ Add the velocity indicator to the populations list: right click on the populations list -> "Show columns" -> "Velocity". Now you can read the current velocity while controling the creature.

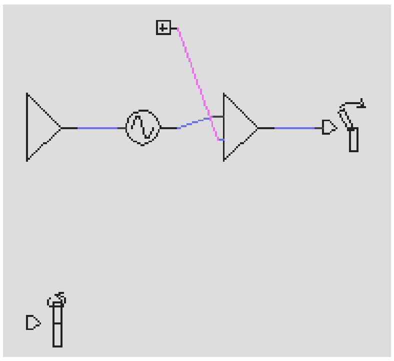
◊ Click the red hold label to release the neuron output.

6. Brainbuilding

Goal: Make the creature move on its own

We will use the "Sin" neuron (sinus generator) to obtain the required pattern.

◊ Connect the new Sin neuron:



X(X,RRL**X** [N] [**Sin**,f0:0.02,-1:0.1] [*] [N,-2:1.5,-1:0.4] [|,-1:1,p:1] [@,p:1]**(X,X,X),X)**

· Explanation:

· [Sin,f0:0.02] is the sinus generator with constant output frequency of 0.02

· -2:1.5 means "connect to the second previous neuron, weight 1.5"

· [*] is a neuron which produces a constant value of one (1).

◊ Build the creature and select it on the list. The creature moves!

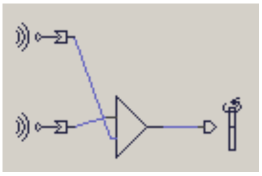
◊ This time the probe will show the varying signal for the generator, neuron and the muscle. Add the first probe to the neuron in front of the muscle. Click and hold the left mouse button on the "1x" label and drag to the right. This will increase the time range of the displayed signal making it more readable.

◊ Add the second probe to the neuron in front of the Sin generator. Move the handle up and down slowly. This way you can control the frequency of the generator, because the Sin generator changes its frequency according to the input signal. Observe the first probe.

◊ **Exercise 1:** Set Experiment/Populations/Creatures: Performance sampling period = 1000. Find the optimal pattern (resulting in the highest velocity). You can change the generator frequency and the constant value (the second input for the neuron in front of the muscle).

◊ Add another probe to the second muscle. Try to make the creature turn left or right by changing the signal.

◊ **Exercise 2:** Make a creature that can turn towards food. You can start with this genotype:



X(X,RRLX [Sin,f0:0.02] [*] [N,-2:1.5,-1:0.4] [|,-1:1,p:1] [@,1:1,p:1] [1:0,2:0](X[S],X,X[S]),X)

The new [S] receptors, placed on the creature's "head", will provide different signal values when the food is close to either side of the creature. Can you use this information to set up the proper weights in the controlling (middle on the picture) neuron? Adjust weights so that the creature does not turn over when it turns!

◊ **Exercise 3:** Using the "Energy" sensory neuron and a genotype from exercise 1, make a creature that moves fast (high frequency of the pattern generator) when has a lot of energy, and slows down gradually until it dies. Then implement the opposite behavior.

7. Communication

Goal: Make the creatures talk to each other

The creatures we built in previous exercises can already sense the environment in one simple way: looking for energy sources. Now it's time for a more dynamic interaction: use specialized neurons to send and receive signals between creatures.

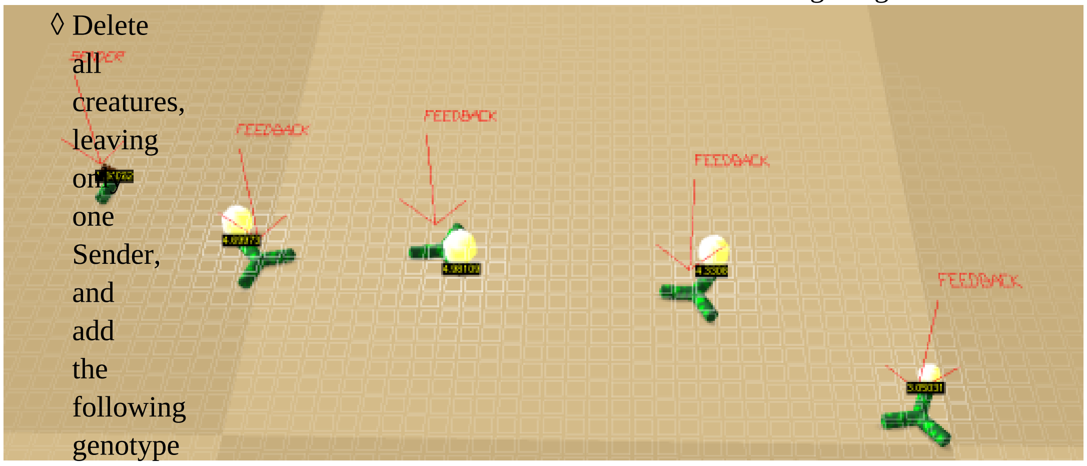
◊ Disable energy calculation or death, set Experiment/Populations/Creatures: Neural net simulation to "Immediate"

◊ Build the following creature (name it *Sender*): **X[Sin][Light,-1:1]** and examine its behavior.
As the name suggests, **Light** is the light-emitting effector. Connecting it to a sine generator creates a periodically flashing device (for positive values of sinus). Note the grid of squares appearing while the light is on – that's the way Framsticks visualizes signal strength.

◊ Add more *Senders*

◊ Try controlling the **Light** effector using a neural probe.

◊ Now let's design the other end – the *Receiver*: **XX[SeeLight][|,-1:1]**
Add several *Receiver* creatures and examine their reaction to light signals.



◊ Delete all creatures, leaving only one Sender, and add the following genotype (called *Feedback*):
X(X,X[*][SeeLight][Light,-1:1,-2:-0.5])
 Place several *Feedback* creatures and one *Sender*, forming a line. Each *Feedback* reacts to incoming light by enabling its own light emitter, but only above the certain activation threshold (depending on the input weight of the ***** neuron). The whole chain of creatures acts like a light conductor, passing the signal between creatures.
Note: You may need to carefully adjust the distance between *Feedback* creatures – bring them closer if there is no reaction, increase distance if their mutual influence is too big.

◊ Make the light signal propagate more slowly. How could this be achieved?

◊ Can you make a cockroach (a creature that avoids light and looks for food when it's dark)?

◊ See the **Fireflies** show (Framsticks Theater). Experiment with the basic Firefly genotype in the Framsticks GUI:
X[SeeLight] [*] [-1:2.26,6:-2,in:0.01,fo:0.01,si:1] [*] [-2:1,-1:-0.5,si:9999,fo:1,in:0] [*] [-2:2,-6:0.3,-1:-0.4,in:0.01,fo:0.01,si:1] [*] [-2:1,-1:-0.5,si:9999,fo:1,in:0] [Light,-1:-1]

◊ Imagine evolution of fireflies. Our goal would be to get creatures that synchronize their flashing despite random starting conditions. How would you define the fitness function? How a creature would be evaluated?

◊ Read the `light.neuro` and `seelight.neuro` files – they implement simple communication neurons we used in this section. Adding more communication channels is as easy as copying these files. See also the **Scripting** section below.

◊ See the **Boids** show (Framsticks Theater) or **Boids** expdef (Framsticks GUI) for more sophisticated inter-creature communication involving scripting.

II. Evolution

1. Evolutionary Optimization

Goal: Let the evolution improve your creatures

In the previous exercises, we tried to make fast moving creatures by hand. Let's see if the evolutionary algorithm in Framsticks can be used for this purpose.

◊ Delete all genotypes and creatures (or restart Framsticks if you previously changed a lot)

◊ Add the following genotype:
X(X,RRL**X** [**Sin,f0**:0.02**]** [*] [**N**,-2:1.5,-1:0.4] [|,-1:1,**p**:1] [**@**,**p**:1]**(X,X,X),X)**
(this genotype is simplified because the Sin generator input is removed)

◊ Set the simulator parameters:

· Experiment/Populations/Creatures: Death: enabled
· Experiment/Populations/Creatures: Neural net simulation: "After stabilization" (and the same setting for Performance calculation)
· Experiment/Populations/Creatures: Performance sampling period: 1000
· Experiment/Parameters: Gene pool capacity: 20
· Experiment/Parameters: Simulated creatures: 1
· Experiment/Parameters/Selection: Crossed over: 0 (why we don't want crossing over?)
· Experiment/Parameters/Fitness: Velocity: 1 (all other: 0)
· Genetics/f1/Morphology: All set to 0 (we don't want to mutate the body)
· Genetics/f1/Neuron net: Add/remove neuron and Add/remove neural connection set to 0, all other: 1

◊ Open a chart. Go to Interface options -> simulator charts, select property "Genotypes: Fitness: Average Fitness", click Apply, click Add chart. Select property "Genotypes: Fitness: Maximal Fitness", click Apply, click Add chart. Close the window (press ESC), and show charts window by pressing [Shift-F6]. Think about the expected shape of these charts, explain why.

◊ Start the simulation. Watch the gene pool window. You can sort the "velocity" column to see if the evolution is making progress. Press [F9] to hide windows and make the evolution run faster ([Shift-F4] will show the window with genotypes). [F4] restores the default layout.

◊ There are many parameters influencing the algorithm. See the "parameters reference" in the Framsticks documentation.

◊ Who is the winner? Is the evolved creature better than your hand-tuned one?

◊ Identify what parts of the original genotype were changed by evolution, and what is the meaning of changes.

◊ Have a look at the charts. Compare average and maximal fitness (speed). Were your expectations correct?

◊ Try to guess what might be the results of evolution if you set the Experiment/Populations/Creatures: Neural net simulation: "Immediate". Then clear all gene pools and populations and test your hypothesis experimentally. Set all parameters appropriately, clear charts, run the evolution, and compare its outcome to the previous results.

2. Tall structures, designed and evolved

   Goal: compare manual design and evolution in the task of building tall creatures

   ◊ Try to build as tall as possible, but stable, creature, using the 'f1' genetic encoding. Do not use neurons, just body. Save its genotype in a file, e.g. "my_tall.gen".
   ◊ Exit Framsticks and launch it again (to ensure all parameters have their default values).
   ◊ Set the simulator parameters:
   · Experiment/Populations/Creatures: Death: enabled
   · Experiment/Populations/Creatures: Neural net simulation: "Disabled"
   · Experiment/Parameters: Gene pool capacity: 20
   · Experiment/Parameters: Simulated creatures: 1
   · Experiment/Parameters/Fitness: Vertical position: 1 (all other: 0)
   ◊ Add a simple X genotype.
   ◊ Start the simulation. Watch the gene pool window. You can add the "Vertical position" column (right click on genotypes list) and sort it to see if the evolution is making progress. Press [F9] to hide windows and make the evolution run faster ([Shift-F4] will show the window with genotypes). [F4] restores the default layout.
   ◊ Who is the winner? Is the evolved creature better than the one you designed?
   ◊ Try to guess what might be the results of evolution if you set the Experiment/Populations/Creatures: Neural net simulation: "Immediate". Then clear all gene pools and populations and test your hypothesis experimentally. Set all parameters appropriately, run the evolution, and compare its outcome to the previous results.

3. Two-criteria optimization

   Goal: discover multi-criteria optimization and related issues

   ◊ Now you will have to evolve creatures that are both tall and simple. Look at the Experiment/Parameters/Fitness settings. You have to maximize "Vertical position" and minimize the number of "Body parts" at the same time.
   ◊ Suggest the fitness function. Think about the values of vertical position and body parts. Enter your weights for these criteria, press apply, and then check the Experiment/Gene pools/Genotypes/Fitness formula.
   ◊ What is the disadvantage of this way of computing fitness?
   ◊ What is "normalization"? Try to guess the meaning of Experiment/Parameters/Fitness/Criteria normalization.
   ◊ Perform the experiment with evolution. Is the result satisfactory? If you have problems with getting interesting results, identify the reasons and ways of overcoming this situation. Repeat the experiment again a few times or compare your result with your friends' results. Who has the best tall-and-simple creature? Consider both criteria and try to compare all best evolved creatures.

III. Additional exercises

1. 'f0' genetic encoding
   ◊ Read the section on the 'f0' encoding from the Framsticks Manual.
   ◊ Build the following shapes: square, cube, tetrahedron, a tent. First draw these shapes on a sheet of paper and specify the 3D coordinates.
   Then suggest some other simple, but interesting shape, and build it.
   ◊ Build a simple neural network with three neurons connected in a line (Touch, N, Bending muscle). Change the placement (embodiment) of the Touch sensor in the

body (assign it to different Parts).

◊ Check the 'f0' genotypes converted from their 'f1' analogues. Enter simple 'f1' genotypes (bodies and also bodies with brains), like the ones from section I.4, press "Apply" each time, and then inspect and understand the "Conversion: f0 genotype" field.

2. Evolution using 'f0'

   ◊ Repeat the experiments II.2 and II.3, this time using the 'f0' genetic encoding. To do it, empty the gene pool and populations, run the simulation and choose the simplest 'f0' genotype from the list of choices. First formulate your expectations as to the differences between 'f1' optimized creatures and 'f0' optimized creatures. Later verify if you were right.

3. Another experiment definitions

   Goal: see how the Framsticks simulator can be controlled by experiment definition scripts

   ◊ Restart Framsticks. Set Experiment/Experiment definition to "neuroanalysis". Press "Apply". Read the Description. Close the parameters window. Load "walking.gen". Open the parameters window and press "Initialize experiment". Close the parameters window and run the simulation. Close the world window to speed up simulation. Do not touch anything until the simulation is automatically stopped. Then inspect some genotypes and have a look at their Genotype/Info field. Discuss how the information computed by this experiment definition can be used and what is the meaning of these values.

   ◊ Restart Framsticks. Set Experiment/Experiment definition to "reproduction". Press "Apply". Read the Description carefully. Press "Initialize experiment". Close the window and run the simulation. Open the Messages window. Watch the simulation.
   1. Can you tell what is the maximal possible number of living creatures?
   2. Is it possible that all creatures die out?
   3. Can you tell how to predict the number of creatures that will appear?
   4. Is the evolutionary improvement taking place in such a setup or not?
   5. What is the difference between this setup and the experiments from section II?

   ◊ Restart Framsticks. Set Experiment/Experiment definition to "learn_food". Press "Apply". Read the Description carefully. Press "Initialize experiment". Close the window and run the simulation. Test the influence of the Experiment/Parameters/Share knowledge parameter on the system. See also Experiment/Parameters/Energy/Food placement parameter.

IV. Scripting

Goal: to learn using FramScript and understand its importance and potential

1. Value types and their conversions

   In the Framsticks GUI open the Console window. Issue the following commands, one by one:

```
? 2+3
? 2.0+3
? 2+3.0
? "a"+2
? ""+2+3
? 2+"12"
? "2"+12
? Math.time
? Math.time%1000
? (0+Math.time)%1000
? (0+Math.time)%1000
```

Try to explain how the results were computed.

Now first guess the result, and then check if you were right:

```
? 2.0*3
? 2*3.0
? 1+2.0*3.0
? 1.0+2*3
? 2*3.0+0.0
? (2*3.0)+0.0
? 2*(3.0+0.0)
? 7+"3"+4
? 7+("3"+4)
? 2*3.5
? 2*3.99
```

Hexadecimal notation can be helpful:

```
? 0xA
? 0xff+10
? 5+"0xf0"
? "0xf0"+5
```

Now meet the arrays (the two last lines are intentionally incorrect):

```
? ["aa",4,"5"][0]
? ["aa",4,"5"][1]
? ["aa",4,"5"][2]
? ["aa",4,"5"][Math.random(3)]
? ["aa",4,"5"][3]
? [aa,4,"5"][0]
```

Now some more values and their types, but introducing variables:

```
var a;  a=5;  Simulator.print(a);
var a;  a=5;  a=a+3.0;  Simulator.print(a);
var a;  a=5;  a=3.0+a;  Simulator.print(a);
var a;  a=5;  a="abc"+a;  Simulator.print(a);
var a=5;  Simulator.print("abc"+(0.0+a));
var a=["aa",4,"5"];  Simulator.print(a[2]+3);
```

Types again, now using the **typeof** operator to check the type of expression in run-time:

```
? typeof(5)
? typeof(3.0)
? typeof(5+3.0)
? typeof(3.0+5)
? typeof("some text")
? typeof("some text"+3)
? typeof(3+"0")
? typeof(null)
```

Comparisons, conversions and detecting **null** ("empty value"):

```
? 4==4.0
? 4=="4"
? 4=="5"
? 4<"5"
? null==0
? null==0.0
? typeof(null)==typeof(0.0)
```

A little puzzle   discover why:

```
? 44<"5"
? "44"<5

? "blahblah"!=0
? 0=="blahblah"
```

Further FramScript information can be found here.

2. Scripting sample

Now have a look at this piece of code. Study every line to ensure you know how it works.

```
World.wrldsiz = 150;                            //make the world large
GenePools.clearGroup(0);                        //delete all genotypes in the first ge
Populations.clearGroup(0);                      //delete all creatures in the first po
Simulator.import("encoding_f1_best.gen", 2); //load genotypes from file
GenePools[0][7].genotype="X";                   //this genotype is missing in the file
GenePools.group = 0;                            //select the first gene pool
Populations.group = 0;                          //select the first population
var i,x,y;
for (i=0; i<30; i++)
{
  GenePools.genotype=i;                         //select i'th genotype (from 0th to 29
  var c=Populations.createFromGenotype();       //construct a creature made from the s
  x = 20+(i%10)*9;                              //compute target coordinates...
  y = 30+(i/10)*20;                             //discover how the values of x and y a
  c.moveAbs(x - c.size_x/2, y - c.size_y/2, c.pos_z);  //and move the center of th
}
```

If you are using Framsticks in version older than 3RC4.8, use the following script instead:

```
World.wrldsiz = 150;                            //make the world large
GenotypeLibrary.clearGroup(0);                  //delete all genotypes in the first g
LiveLibrary.clearGroup(0);                      //delete all creatures in the first p
Simulator.import("encoding_f1_best.gen", 2); //load genotypes from file
GenotypeLibrary.getGroup(0).getGenotype(7).genotype="X";  //this genotype is missi
GenotypeLibrary.group = 0;                      //select the first gene pool
LiveLibrary.group = 0;                          //select the first population
var i,x,y;
for (i=0; i<30; i++)
{
  GenotypeLibrary.genotype=i;                   //select i'th genotype (from 0th to 2
  var c=LiveLibrary.createFromGenotype();       //and construct a creature made from
  x = 20+(i%10)*9;                              //compute target coordinates...
  y = 30+(i/10)*20;                             //discover how the values of x and y
  c.moveAbs(x - c.size_x/2, y - c.size_y/2, c.pos_z);  //and move the center of th
}
```

If you discovered how the coordinates are computed based on the number of creature (varialble i), now ensure that the simulation is stopped, copy this piece of code, paste it into the console window, and execute it. See how the creatures are located: make the world window full screen and rotate the world.

As you can see, this piece of code automates many operations.

3. Your own fitness functions

   Have a look at the Experiment/Gene pools/Genotypes/Fitness formula. This formula is automatically updated each time you change weights for the optimization criteria. However, you can create the fitness formula according to your wishes using **if ... then**, temporary **var**iables, loops, etc. First you have to learn about the fields you can use in the fitness formula. In the Framsticks GUI, choose menu "Help" and "FramScript: reference". Check the "Fitness formula" context and the "this" class, its methods and values ("attributes").

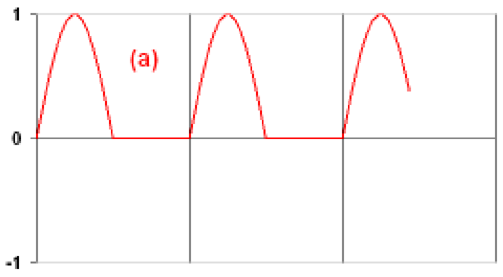
   Then see the "Math" class in the global context. Now is the time for your experiments. Implement the following fitness functions, and for each function, explain its meaning and verify it practically. Take care of special situations (divisions by zero, square roots of negative values, etc.).

   1. Number of Joints + 0.01 * number of Neurons
   2. Square root of velocity
   3. (Number of Parts + number of Neurons) divided by the number of neural connections
   4. Conditional fitness: if a creature has less than 5 Parts, then fitness is creature height. Else fitness is creature height diminished by 0.1*(number of Parts minus 5).
   5. (advanced task) Fitness is the number of Joints divided by the volume (size_x*size_y*size_z) of the Model made from Geno.

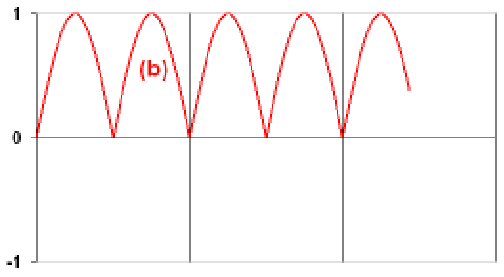
   See more scripting resources including the full reference for contexts, classes and methods.

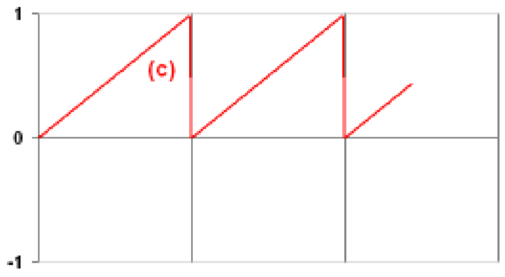
4. Your own neuron classes
   1. If you are going to write larger pieces of code in FramScript, consider using Framclipse.
   2. Find the "scripts" subdirectory within the Framsticks installation directory. Have a look at file extensions and their contents. Then view the "threshold.neuro" file. The *.neuro files have three parts: preamble (basic neuron description), code (delimited by the ~ signs) and properties (neuron parameters). Look at the file carefully and discover how exactly the threshold neuron works. Then run Framsticks and use the genotype
      X[*][**Thr**,-1:1,**t**:-0.2,**hi**:0.3,**lo**:0.8]
      to test it.
   3. Now have a look at the source of the noisy neuron (the noisy.neuro file). Can you explain how it works? (a brief description of this neuron is also available in the Framsticks chapter of the book Artificial Life Models in Software). Find out what genotype to use to test this neuron. Then verify if your explanation was correct.
   4. Discover what procedures ("methods") can be used in FramScript. In the Framsticks GUI, choose menu "Help" and "FramScript: reference". Take some time to browse various classes that come to your attention. Check out the "Math" class, its methods and values ("attributes").
   5. Copy or save the noisy.neuro under the different name (exa.neuro) but in the same directory. Edit this file and change the neuron name by modifying the appropriate line: "name:ExA" (neuron names must start with a capital letter). Long name should be "Exercise A". Now try to modify the function go() so that the output of the neuron is the sinus function. Save the file, restart Framsticks, see the Messages window for errors. If everything is fine, create a genotype to test your ExA neuron. If your neuron code was invalid, try to correct it or ask the tutor.

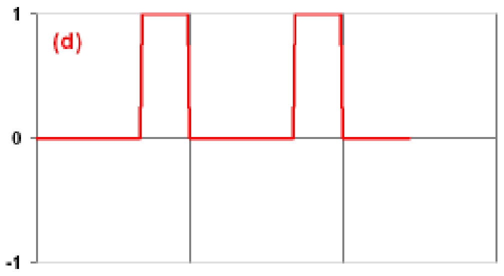
6. Now there comes the real exercise (a). You have to design the neuron which outputs the signal as shown on the figure. Note that your ExA neuron should have one parameter (t) to set the period of this pattern (the number of simulation steps after which the pattern repeats). Instead of restarting Framsticks each time to load your new neuron code, try clicking the Simulation Parameters/User scripts/Reload neuron definitions.



7. If you were successful, solve the next three tasks. For each of them, create a separate file (exb.neuro, exc.neuro, exd.neuro). In Exercise (d), you need two parameters which specify the widths (in time steps) of the high and low output states. For some tasks, you may need to introduce the init() function to initialize some Fields. Ask the tutor if you cannot cope with some problem, or skip this problem if you cannot get help.







8. There are five more challenges for you! First, build a neuron which makes the input signal smooth by averaging the current input with the two previous inputs. Use weighted input sum, not individual inputs.
9. Then build a neuron which outputs the maximum value of its inputs. Design the appropriate neural network to test it.
10. Then build a neuron which multiplies the first input by the second one, and outputs the result.
11. Then build a memory cell. The first input is the value to save. The second input is the control signal: if it is less than zero, then your neuron should memorize the first input value (and output it). If the control signal is zero or more, your neuron should output the memorized value regardless of the first input.
12. Finally, build a neuron which outputs *two* values at the same time using channels. Your neuron should compute weighted input sum and output it as (always) positive value in the first channel, and (always) negative in the second. Later you can use the ChSel neurons to select a single channel from a multichannel output.

Note that although we designed regular neurons here, the go() function can use information about the world and other creatures, so you can easily design your own sensors. Effectors can be built as well, but in most cases you need to modify the experiment definition script to make them work properly.

5. Your own macros

◊ Find the "scripts" subdirectory within the Framsticks installation directory. Have a look at the contents of files with extension ".script". They are similar to the *.neuro files, right? And also have the "code:" section 😉

◊ Now analyze carefully and ensure you understand "neuroclsreport.script", "foodcircle.script", "creaturescircle.script" and "gallery.script". In the Framsticks GUI, find the Simulation Parameters/User scripts. You can run the scripts to see how they work.

◊ Remember the sample in section IV.2? It can be saved in a *.script file and executed with just one click.

◊ If you want to experiment and modify some .script file, first copy it with a different name. Then modify the source and restart the GUI to test your changes. Alternatively, you can test your macros by pasting their source to the console window. When you are done, implement the following macros:

1. arrange all living creatures (from population #0) in a square (and issue a message when there are no creatures to arrange)
2. as above, but the tallest creature should stand in the middle
3. arrange them in a line, according to their height
4. increase the water level in the World by +0.5
5. print a report   display the number of objects (genotypes, creatures) in all gene pools and populations, and the names of gene pools and populations
6. for each genotype, add 10 mutants to the gene pool. The original genotypes should remain unchanged, and mutants should have numbers appended to their original names
7. jiggle creatues! for all creature groups, for each living creature, for all its MechPart's, add a small random number to its velocity. It is a good idea to put the jiggling function in the experiment definition, and run it e.g. every 200 simulation steps.
8. (advanced task) as above, but for all MechParts with the Touch neuron, add a small constant number to MechPart.vz

◊ If, for some reason, you would like to save some data in a regular file during simulation, you can use the File object. See the `scripts\neurof0html.script`, no need to understand in detail how it works, but it illustrates how the File object can be used. You can run this script and see the file created in the *scripts_output* subdirectory.

6. Command-line interface (CLI)

Goal: meet the Framsticks CLI and learn how to use it for automated experiments

1. Download the CLI and prepare it for use.
2. Run "frams". Enter "lm". Inspect the "frams.ini" file, pay attention to the init() function and learn how macros are associated with implementing functions.
3. Experiment with the following macros: im, sa, st, lg, lc, qu.
4. See that you can enter commands as in IV.1.
5. load "walking.gen", set the gene pool capacity to 20 (using FramScript), run the simulation until 100 creatures are evaluated (using FramScript **while** loop), and save results to "walk2.gen".
6. Create a "cmds.txt" file with the commands you issued in the recent exercise, quit frams, and use it in batch mode. Run the Windows (or Linux) command-line interpreter (cmd.exe or sh), and enter:

```
frams < cmds.txt
```

   7. Learn how to use parameters in the command line   enter

      ```
      frams --help
      ```

      and then, for example,

      ```
      frams "? 2+3" "lo walking.gen" "lg" "qu"
      ```
   8. Make neuroanalysis.expdef evaluate all genotypes from walking.gen, and then save the resulting gene pool ("sa walk-analyzed.gen"). If you use an experiment definition other than *standard.expdef*, you have to set it as an argument in command-line like this: `frams "ex neuroanalysis"`. Don't forget about initializing the experiment.
   9. Create a "cmds-neu.txt" file with the commands you issued in the recent exercise, quit frams, and use it in batch mode.
7. Your own experiment definition

   See the scripts/*.expdef files, choose some simple expdef, discover what "onSomething" events are implemented, save the expdef with a different file name, modify it, restart the GUI, and test your modifications. Have a look at simple `learn_food.expdef` and `generational.expdef`. If you successfully completed previous tasks, you are now able to implement your own definitions of experiments. Some ideas for experiment definitions to create:

   1. Compute Genotype's distance as a difference between the place of birth and the place of death.
   2. Compute fitness as the height of the topmost Part, averaged during the life span (measured in onUpdate events).
   3. *Cleaning*: when two creatures collide outside of the world center, move the bigger one to the world center. If this place is already occupied (collision with another object, use `LiveLibrary.creatBBCollisions`), the creature is moved higher.
   4. *Fight for resources (arena)*: creatures are born on two sides of a spot and the one which is able to cover most of the spot gets higher fitness.
   5. *Pursuit and evasion*: two gene pools, two contradictory fitness functions, one group of creatures is evolved to escape, the other one   to follow them.
8. Your own Framsticks Theater show

   See the scripts/*.show files, choose some simple show, discover what "onSomething" events are implemented, save the show with a different file name, modify it, restart the theater application, and test your modifications. You have to use a registered version of the theater to test your new shows.
9. Your own OpenGL style

   Have a look at the 3dobj/*.style files. Proceed similarly as described above. Read the tutorial on scripting, it includes a related example. You need a registered Framsticks GUI to test your new OpenGL styles. See also the Spooksticks story.

25 If you have questions

If you have questions or problems that were not resolved by available documentation, help, and publications, have a look at the on-line forums, where Framsticks users can share ideas and help each other. Before asking a question, please check the documentation.

Or you may want to contact Framsticks authors.