Framsticks Deathmatch Challenge

Tutorial v1.0

Walter de Back
Virtual Life lab
Institute for Computing Sciences
Utrecht University, NL

www.aisland.org/vll
vll@aisland.org

Maciej Komosinski
Institute for Computing Sciences
Poznan University of Technology, PL

www.frams.alife.pl

Contents

1 Introduction

This document describes the *Framsticks Deathmatch v3.0*, an educational tool created to illustrate some of the problems in evolutionary computing, evolutionary robotics, and artificial life. It is intended for use in practical courses in these fields. The *Framsticks Deathmatch* experiment also introduces key features of the Framsticks simulator (**Komosinki & Ulatowski, 1997**).

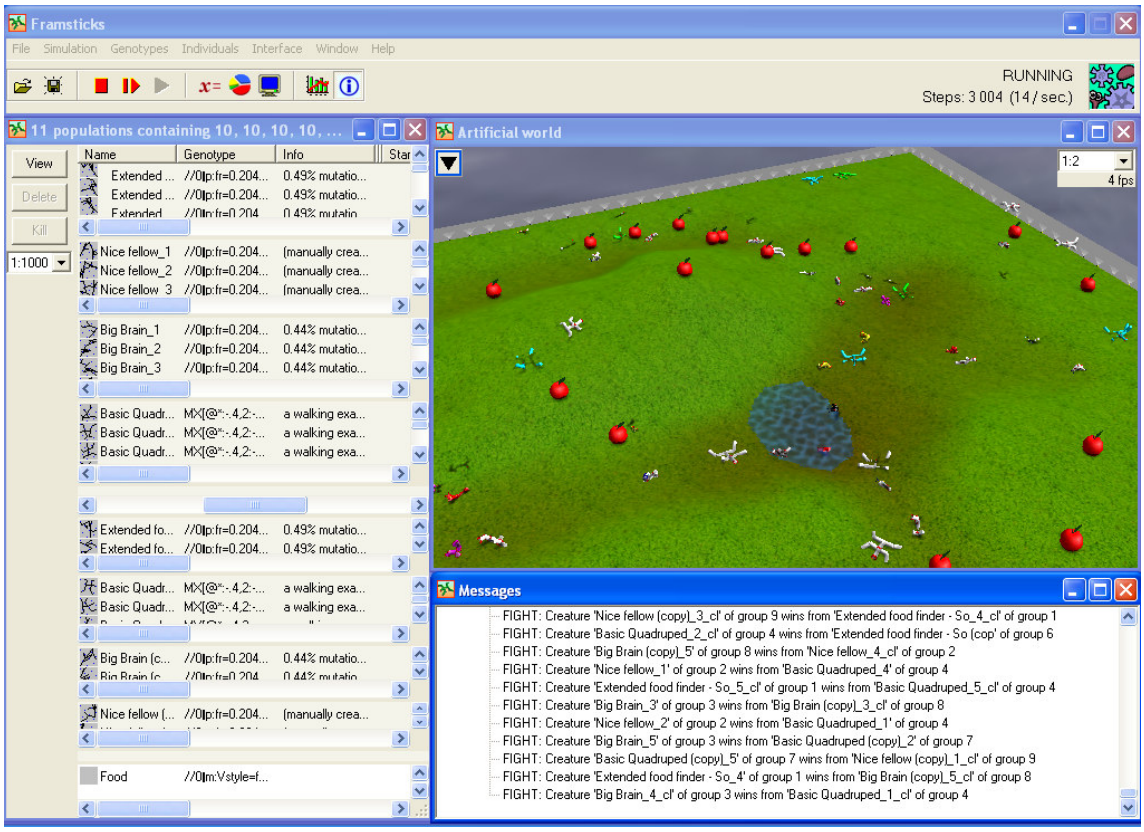
The Deatchmatch is developed by the Virtual Life lab, Utrecht University, in cooperation with the developers of the Framsticks simulator, Maciej Komosinski and Szymon Ulatowski, Poznan University of Technology.

In this document, the Framsticks Graphical User Interface (GUI) for Microsoft Windows is described, although there are other possibilities. In particular, command-line interfaces can be used for long-term experiments (they are faster than the GUI), and remote GUIs can be used to connect to Framsticks server(s). Framsticks software for Linux can also be used. Refer to the Framsticks website for available downloads.

2 What is the Deathmatch

The Framsticks Deathmatch is a tournament between teams of creatures, as well as between teams of students.

A class of students is divided into several groups. The task of each group gets is to develop a creature to participate in the Deathmatch. The methods you can employ in order to develop your own creature are virtually unrestricted. Through this you will learn about a wide field of scientific problems and methods concerning embodiment, neural networks, and artificial evolution.



Picture 1. The Framsticks Deathmatch in the Windows GUI

The Deathmatch works as follows:

Start

To start the Deathmatch, each student groups hands in one genotype of the creature they developed. This single creature is copied several times to constitute a creature *team*. Each team gets a unique population as is shown in the right-hand panel in the screenshot (Picture 1). There is one extra population for food items.

Goal

The goal for a team is to stay alive longer than the other teams. This goal can be achieved in two ways: (1) driving other teams to extinction by *fighting*, or (2) extending your team by *cloning*.

Creatures stay alive by maintaining a positive energy level through eating food and fighting other teams.

Losing energy

When started, the creatures get a starting energy level and begin to crawl around in the world. Creatures lose energy at every step (*idle metabolism*[1]), when they move (*muscle dynamic work*), and when they resist an external force – when they run into something (*muscle static work*).

It is also possible to transfer a portion of energy from the parents to the child when cloning (*transfer energy)*.

Winning energy

To stay alive or reproduce, creatures must collect energy. Creatures can do several things to collect energy:

- **Forage**: when colliding with a food item, the energy of the food item is transferred to the creature (*food's energy*)
- **Fight**: when colliding with a creature of another population, a fight takes place. The winner and loser of the fight are determined by comparison of their energy levels (the one with most energy wins). A portion of energy (*win energy*) is transferred from the loser to the winner.

Teams

The number of populations/teams (*number teams*) and size (*creatures in team*) are instantiated at initialization. The team sizes are not static, however. They vary throughout the Deathmatch. The goal of the match is to stay alive as long as possible, both the individual creatures and the team as a whole. The team size is influenced by the creatures abilities to:

- **Clone**: when a creature reaches a certain level of energy (*reproduction energy*), this creature is cloned and put somewhere in the world near its parent.
- **Die**: when a creature has zero (or negative) energy, it dies and is removed from the world. Its lifespan is added to the team score.

Score

The score of a creature is determined by its lifespan. The score of a team is the sum of the lifespans of all creatures in the team. The winning team of a match can be awarded with an additional bonus (*bonus*).

The match can be repeated several to get average out contingencies of the world interactions. This can be done automatically (*restart after extinction* with *number of matches*) or manually. If you re-initialize after a match, the scores are preserved and the new scores are added to the last ones.

3 Creature development

We strongly encourage the free-forming of creatures (various methods of creature manipulation were used for example by **Mandik, 2000**). This means you can use *any method available* to develop their creature. This means you have to explore the system and in this process encounter the various features of Framsticks, and the wide possibilities for the evolution of morphology and neural controllers. Four major methods can be divided:

- **Using existing creatures**: The Framsticks simulator comes with various walking, swimming and foodfinding creatures. Moreover, the Framsticks Experimentation Center[2] offers a continuously increasing pool of different genotypes for creatures.

  This is not cheating: Since all student groups can use the available genotypes, the further development of these creatures is necessary. Moreover, this encourages creative development by seeing many examples.

- **Editing creatures**: A stand-alone creature editing program is available at the Framsticks website. Students can use this FRamsticks Editor (Java application), FRED[3], to easily edit existing creatures or construct their own without the need for detailed knowledge of genetic encodings. Download FRED v2.0 at : http://www.frams.alife.pl/common/dl/Fred2.jar

[1] Terms in *italic* refer to the parameters, see section 6.

[2] Check the Framsticks Experimentation Center at http://www.alife.pl/fec/www/index.php

[3] More information about FRED at http://www.frams.alife.pl/dev/index.html

- **Genetic coding/editing**: Framsticks offers immediate preview of genotype encoding. When you change a genotype, you can see the changes immediately in the body and brain structure of the creature. You can even select genes to see their counterparts in the creature, and the other way round. There are a few genetic 'formats' available; ***f0***, ***f1*** and ***f4*** are the most common. With ***f1***, it is possible to quickly design basic body concepts, while ***f0*** is quite simple and has the biggest abilities (it is the basic low-level encoding). Although coding in various genetic formats seems difficult at first, it is pretty easy once you try it. Especially the ***f0*** encoding is easily understood and adapted for specific purposes. For a comparison between the various genetic encodings in Framsticks see (**Komosinski, 2002**).

  You can use all genetic formats in the Deathmatch.

- **Fitness optimisation**: Optimising creatures is required in any case, since setting the parameters of the neural controller is almost impossible to do by hand. The standard experiment definition offers many possibilities to optimise creatures' morphology and neural controller.

  This requires students to experiment with many aspects of exogenous (i.e. fitness-based) evolutionary computation:

  - fitness criteria: e.g. distance, velocity, lifespan, etc.
  - genetic operations and their probabilities: crossover, mutation
    - applying it selectively to aspects of morphology or neural controllers
  - selection mechanisms: random, roulette wheel, various tournament-sizes

  Moreover, creatures must be prepared to the endogenous (not fitness-based) conditions of the Deathmatch, which requires experimentation with:

  - a number of simultaneously simulated creatures: interaction between (populations of) creatures
  - varying energy schemes: starting energy, metabolism, muscle work costs, food energy, etc.

A pragmatic combination of the methods above is what we mean by free-forming. It introduces you to many complex issues of evolutionary computation and artificial life in a playful and competitive way. An important feature of the Deathmatch in this process is the difference between the exogenous fashion in which creature are developed, and the endogenous environment in which they will be tested and compete. This can help clarify a key difference between, on the one hand, the application of standard evolutionary computation to biological problems and, on the other hand, biology-inspired evolution in artificial life systems (such as in Ray's Tierra, and Yaeger's PolyWorld).

Besides using the free-forming methods, more advanced students can adjust the experiment itself to get even better results. This introduces them to the potential of using Framsticks in research projects:

- **Script writing**: In some cases it is advisable to change the experiment definition. For example: An integral part of the Deathmatch is the battle between many populations. Preparing an advanced creature for this task can be done by subjecting it to a co-evolutionary process. This can be set in the experiment definition. It is written in an imperative (C and JavaScript-like) scripting language called *FramScript*[4]. To learn *FramScript*, take a look at some examples in the script_sample folder, or look at the standard.expdef or deathmatch.expdef. More experiment definitions will come available at the Experimentation Center.

- **New neurons**: Another application of *FramScript* is defining new neurons, or receptor neurons. For example: For the Deathmatch, it could be useful to define receptors that 'smell' the energy of creatures and food in other populations, but not in its own population[5]. This prevents creatures in the same team to be attracted to each other, while they should be foraging or fighting other creatures.

For a comprehensive example in which all these methods are used, see section 6.

[4] The reference for FramScript is available at http://www.frams.alife.pl/common/script/docs/index.html

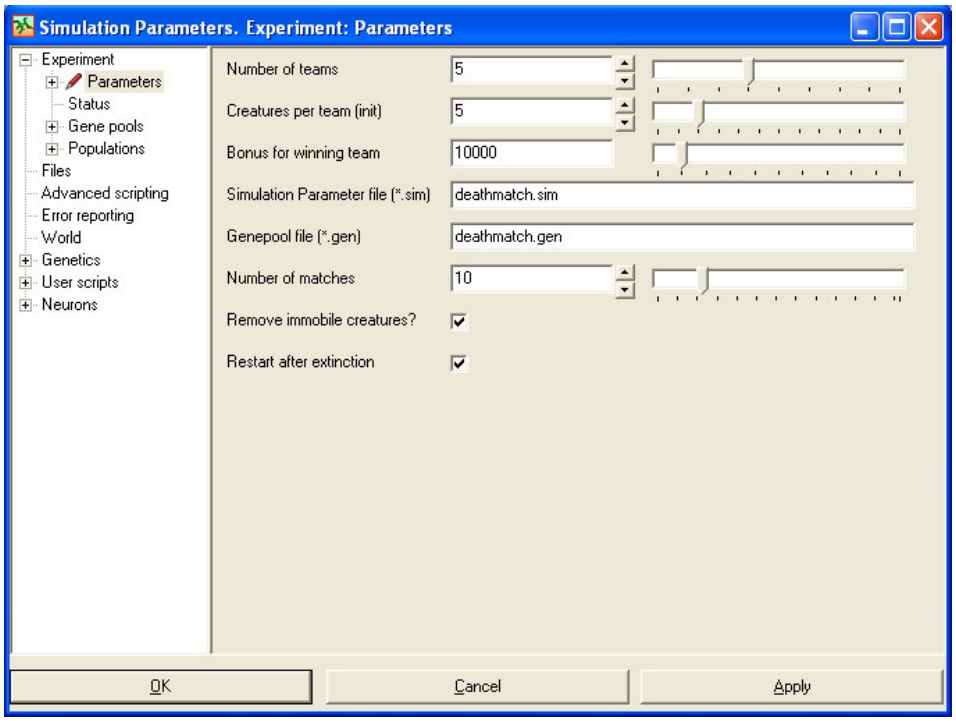
[5] For a sample of such a receptor neuron defined in FramScript, see http://www.aisland.org/vll/projects/smell_others.neuron

4 Parameters

The deathmatch experiment definition has its own specific parameters which appear in the Parameters window (shown on Picture 2). This section provides short descriptions of the most important parameters specific to Deathmatch, and gives names of their counterparts in the experiment definition.

You can load the Deathmatch in the parameter window by selecting Deathmatch_v3 and pressing apply (or reload). After loading, the Deathmatch-specific parameters appear in the parameters window.

The Deathmatch parameters are divided into several groups by events during the lifetime of a creature: Birth, Life, Foraging, Fighting, Cloning.



Picture 2. Deathmatch parameters window.

Place	Name	Description	Name in deathmatch.expdef
Experiment		Title *Deathmatch_v3* appears. Displays short description of deathmatch.	

	Initialize	(Re-)Initializes the experiment.
		The team scores are not set to zero.
			`OnInitExp()`
			`global scoreA`
			`global scoreB, etc.`

	Reload	Reloads the experiment definition. Builds number of teams. Reloading *resets team scores* to zero!	`OnExpLoad()`
Parameters	Number of teams	Number of populations to be filled with genotypes at initialization. (default = 5)	`ExpParams.numberTeams`
	Creatures in team	Number of creatures (clones) the teams to be filled with *at initialization*. The number of creatures varies during the match. (default = 5)	`ExpParams.numberCreatures`

	Bonus for winning team	The last surviving team, the winning team, gets this amount of 'lifesteps' as a bonus. (default = 10000)	`ExpParams.bonus`
	Simulation parameters	Sim-file loaded at reload and initialisation. Deteremines all parameters setting in this window. After changing setting, save them (Ctrl-F2) (default = deathmatch.sim)	`ExpParams.paramfile`
	Genepool file	Gen-file loaded at initialisation, which contains the particiapating creatures. (Number of creatures in file >= number of teams) (deafulat = deathmatch.gen)	`ExpParams.genepoolfile`
	Number of matches	Total number of matches simulated after each other (Restart after extinction must be checked!)	`ExpParams.numberMatches`
	Remove immobile creatures	Automatically remove immobile creatures each 100 steps.	`ExpParams.removeImmobile`
	Restart after extinction	Automatically restart after all teams (except winning team) are extinct	`ExpParams.autorestart`
Birth	Creation height	Determines the height creatures are born. (default = 10)	`ExpParams.creath`
	Prevent collision on birth	If checked, new creatures are placed next to parent.	`ExpParams.clonenocontact`
	Starting energy per stick	Starting energy is multiplied by number of sticks. **Important** if creatures with different morphologies are used! Also influences fight handling (winner is determined by energy relative to number of sticks).	`ExpParams.energyperstick`
	Starting energy first generation	Creatures of first generation start out with this level of energy. (default = 10000)	`ExpParams.Energy0`
	Starting energy clones	Clone creatures start out with this level of energy (added to inherited energy)	`ExpParams.BaseEnergyClone`
Life	Idle metabolism	Energy every stick of a creature consumes every step. (deafult = 0.01) **Heavily influences duration of match!**	`ExpParams.e_meta`
Foraging	Feeding rate	Number of food items in the world. Every time a food item is consumed, one is created at a random place in the world. (default = 25)	`ExpParams.feed`
	Food's energy	Energy of every food item. (default = 250)	`ExpParams.feede0`
	Food's genotype	Genotype of food parts can be entered here. (default = "//0\nm:Vstyle=food\np:")	`ExpParams.foodgen`

Fighting	Win energy	Energy transferred from the loser to the winner of a fight (default = 500)	`ExpParams.winenergy`
	Wait after fight	After a fight, creatures cannot engage in a new fight for this amount of steps. This prevents continuous fights and energy transfers (the draining of a loser's energy) (default = 150)	`ExpParams.waitFight`
Cloning	Reproduction energy	Creatures can clone if their energy level exceeds this level, if the limit teamsize is not exceeded. (default = 11000)	`ExpParams.reprEnergy`
	Reproduction age	Creatures can clone if their lifespan (in steps) exceeds this level, if the limit teamsize is not exceeded. (default = 2000)	`ExpParams.reprLifespan`
	Inherit energy	Child inherits a proportion of energy from parent. Is added to clone base energy. (default = 0.3)	`ExpParams.reprInheritEnergy`
	Wait after cloning	After cloning, the parent cannot clone agin for this amount of steps (default = 150)	`ExpParams.waitCloning`
	Limit teamsize	Population / Team sizes cannot exceed this number. (deafult = 20)	`ExpParams.poplimit`

Several parameters, which are not Deathmatch specific are important as well.

Population → TeamA	Muscle static work	Energy consumption of a muscle resisting an external force each step. (default = 0.005)	`CreaturesGroup.em_stat`
	Muscle dynamic work	Energy consumption of a muscle moving a stick each step (default = 0.001)	`CreaturesGroup.em_dyn`
Genetics		There is **no evolution** in the Deathmatch, only cloning. All genetic probabilities are turned off.	
	f1 → excluded modifiers	The 'e' and 'E' modifiers are excluded, because they influence the energy of creatures.	
World	Type	Height field	
	Size	100	`World.wrldsiz`
	Map	r 30 30 4	`World.wrldmap`, `WorldMap.*`
	Water level	-2.5 There is a small pool in the world	`World.wrldwat`
	Boundaries	Teleport	

5 Experiment definition

Experiment definitions in Framsticks are event-based. This section provides a short description about what happens at the different events and different functions in experiment definition.

Event	Activated at / Action	expdef function
Experiment Load	*'apply' or 'reload'*	`onExpDefLoad()`
	1. Variables (see section 6) are declared and instantiated 2. Simulation file is loaded (to determine number of teams to create) 3. Populations are created	
Experiment Initialization	'*initialisation'* and '*autorestart'*	`onExpInit()`
	1. Populations are emptied 2. Simulation file is loaded 3. Genotype file is loaded 4. Populations are filled: Genotypes are cloned *Number of teams* are filled with *Creatures in team* And placed randomly in the world	
Step	*Every step*	`onStep()`
	1. For every individual calculate idle metabolism (including aging) 2. Check whether number of food items is equal to preset number of food items	
Add food	*By onStep*	`addfood()`
	- Places food items with food energy randomly in world	
Creature step	*By teamStep-function, e.g. onTeamAStep()*	`onCreatureStep()`
	- If a creature has sufficient energy and lifespan, clone it and place the child somewhere near the parent	
Food Collision	*Whenever a food item is hit*	`onFoodCollision()`
	1. Determine which one of the colliding creatures is the food item 2. Transfer energy from the food item to the real creature	
Creature Collision	*Whenever a creature is hit. This function is indirectly called by population-specific functions, e.g. onTeamACollision()*	`onCreatureCollision()`
	1. Check if they are both real creatures (and not food) 2. Check if they did not very recently collide 3. Determine winner of fight 4. Transfer energy from loser to winner 5. Make note of the fact that these creature recently collided	

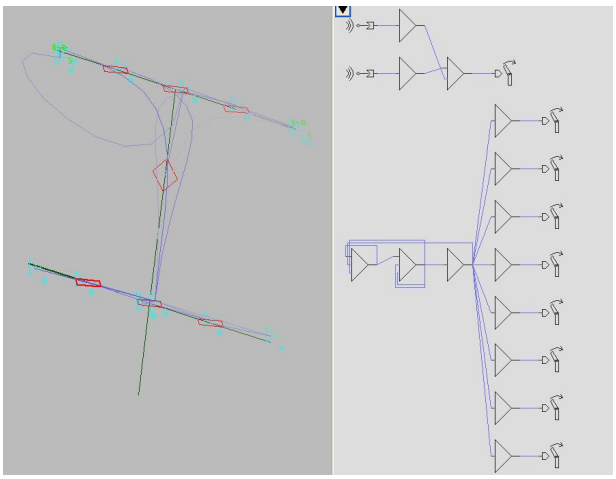
Death of creature	*Whenever a creature has zero or negative energy* 1. Add the lifespan of the creature to the score of its team 2. If all creatures in population are dead, give a message which team has gone extinct	`onXKill(),onDeath()`
Extinction of all teams	*When every population has gone extinct* 1. Give a message about which team is the winner 2. Give the statistics 3. Restart (re-initialize) experiment if autorestart is turned on.	`onExtinct()`

6 Example creature development

This section provides an example of the development of a creature for the Deathmatch challenge. We are going to free-form a creature that should meet the task. All available methods will be employed.

1. **Study the Deathmatch.** First, we will need to study the Deathmatch experiment itself. It can be seen that our creature should be able to (1) move around, (2) find food, and (3) collide with other creatures. We could choose to make construct and evolve a walking creature. This is, however, not a trivial task. Creating a walking creature is difficult, let alone one that finds food (one that has chemotaxic behaviour). Besides, the other student teams probably use one of the existing creatures as well, so why won't you.
2. **Browse existing creatures.** Let us choose one of the existing creatures. Load for example the creatures in *walking.gen* and/or *demo-chase.gen*, turn off all the genetic operations (mutation and cross-over to zero), and calculate the fitness values (distance or velocity). Choose one of the creatures that seems to perform well.

   Let's say I like the 'foodfinder' creature, and use this to further develop my deathmatch-creature.



Picture 3. The foodfinder creature, body and brain.

This creature is already capable of walking and chemotaxis. This solves our major problems. We cannot just submit this creature, since all other student groups have seen this creature as well, and could choose to use it. We will have to develop this creature further, in order to win the Deathmatch. We should analyse how this creature works, and make it better.

3. **Think about how to further develop the creature.** The brain of this creature consists of a central pattern generator, which makes it walk. We see the clever pattern generator at the bottom-right in the screenshot (Picture 3). We can use this part, without much change.

   In the upper-right corner, we see the neural structure that is responsible for the chemotaxis. There are two 'smell' receptor neurons (detecting all energy sources), connected as input to a perceptron, and the output is signalled to the bend muscle neuron on the middle of the body. The neural connections are already optimised through evolution as to exhibit the following signalling: If the activation of the left smell neuron is higher than the activation of the right smell neuron, the bend muscle is activated to turn to the left[6]. This performs very well (prolongs lifespan) in an environment in which all energy sources are eatable – for example, a world in which there is only one creature, and many food items. The environment in which our creature must be able to perform (in the Deathmatch) is quite different, however. There are many energy sources (food items and many creatures), of which some can provide us with energy, and some can't. Creatures from other populations can give us energy only if we win the fight. Food items always give us energy. But creatures from our own team can't give us energy, and it would thus be a waste of time and energy to move towards them. We should adapt the foodfinder to behave in a sensible manner in these respects.

4. **Using FRED.** One idea could be to provide the chemotaxis structure of the brain with an energy level detector neuron, which detects the energy of the creature itself. That is, it starts out activated as 1 (starting energy) and drops according to the lowering energy level. When energy is consumed or won, it could be activated as > 1. This would, in principle, enable the creature to 'decide' whether it should hunt for other creatures and fight (winning a lot of energy) or find some food (to stay alive). Somewhat more accurately phrased: the own energy level could influence the chemotaxic behaviour in a sensible way[7].

   We can choose to use FRED, the Framsticks Editor, to add the energy detector neuron. However, Fred uses ***f0*** genotype format, while the foodfinder is encoded in the ***f1*** format.

   f1 genotype of the foodfinder:

   lllfffX[0:2.420,2:-2,1:-1][-1:1,0:1,0:-1][-1:1](RRIIIfffMMMX[|-1:-10]IIIFFFMMMX[|-2:-1],fffIXIIIfffMMMsX[|6:10,3:-10](RRIIIfffMMMIX[|-4:-10]IIIFFFMMMIX[|-5:-1][S:1],,RRIIIfffMMMIX[|-7:10]IIIFFFMMMIX[|-8:1][S:1]),RRIIIfffMMM X[|-10:10]IIIFFFMMMX[|-11:-1.784])

   This problem is overcome easily: If you double-click the genotype, the genotype data window pops up. Under 'Conversions', we see the conversion to ***f0***, which consists of a prefix (//0) and the lists of parts, joints, neurons, and connections respectively:

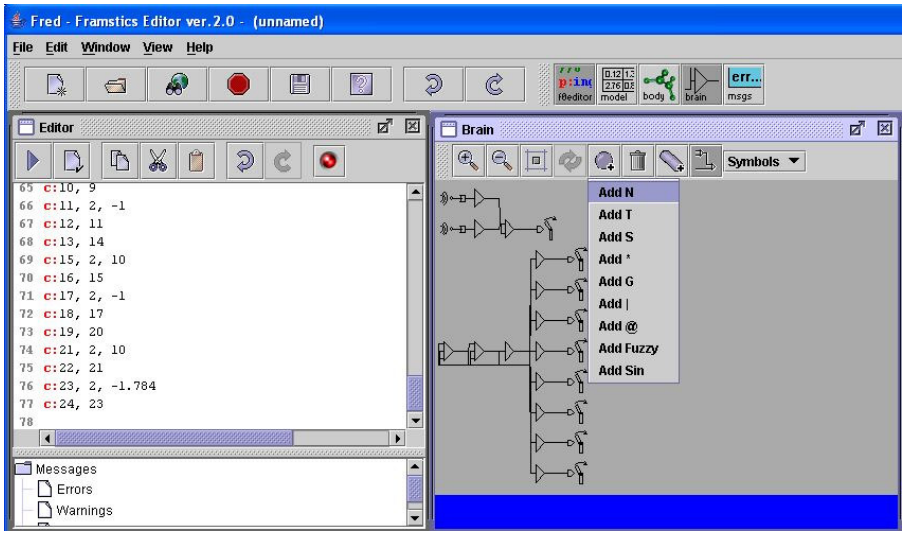
```
//0
p:fr=0.2048
p:0.55981, m=4, fr=0.2048
p:0.559954, -0.484317, m=2, fr=0.124846, ing=0.0892857, as=0.0892857
p:0.560093, -0.955687, fr=2.0441, ing=0.0647866, as=0.0647866
…etc…
j:0, 1, dx=0.55981
j:1, 2, rx=1.5706, rz=-1.5705, dx=0.484317, stam=0.0892857
j:2, 3, dx=0.47137, stam=0.0647866
j:1, 4, dx=0.779905, stam=0.15625
…etc…
n:p=1
n:p=1
n:p=1
n:p=2
n:j=1, d="|:p=0.732143,r=0.333333"
n:p=3
n:j=2, d="|:p=0.80564,r=1"
…etc…
c:0, 0, 2.42
c:0, 2, -2
```

[6] It is comparable to the Braitenberg vehicle type 2a (see "Vehicles: Experiments in Synthetic Psychology", Valentino Braitenberg, 1986).

[7] Of course, much more is needed for this then just adding an energy detector. For example, extending the neural structure with a hidden layer, and some means to discriminate food from other creatures.

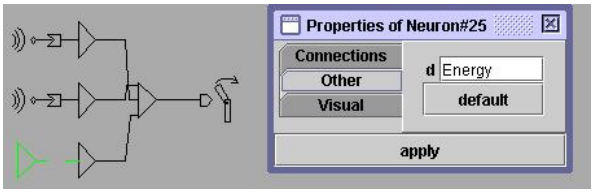
```
c:0, 1, -1
c:1, 0
c:1, 1
c:1, 1, -1
…etc…
```

Select and copy this genotype, and start Fred (by double-clicking the Fred20b.jar file). In the editor window, paste the genotype, and build the model. Start the brain window, and the brain structure will appear.



Picture 4. Fred snapshot, genotype and brain shown.

Now, you can add a neuron, selected from the drop-down menu. Your new neuron will appear as a standard neuron, without any place on the body, and without connections. A right mouse click makes a neuron properties panel pop up. In this panel, you can (1) select a part of the body on which the neuron should be, (2) define its type, i.e. "Energy". Now connect it to another new standard neuron, and make this neuron connect to the neuron which connects to the bend muscle.



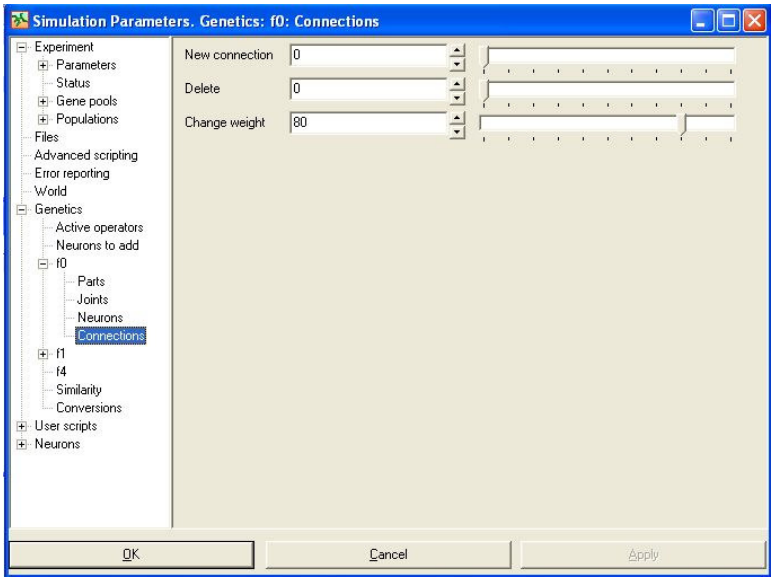
Picture 5. Neuron properties panel in Fred.

The genotype is automatically changed, and now incorporates the energy detector neuron. You can copy the genotype from the Fred editor window, and paste it into a new genotype in Framsticks.

It should be mentioned that adapting ***f0*** genotypes is easily done in the Framsticks as well. Fred is not really necessary for this (although it is a handy tool if you want to construct complex creatures from the start).

5. **Reconfiguring neural connections by fitness-optimising evolution.** We now have a new creature brain, with new connections. These new connections are not configured yet, and would probably disturb the smooth chemotaxic behaviour it had before. To reconfigure the connection weights, we should subject it to evolution.

   We do not want evolution to change the structure of the body, nor the brain. We only want it to re-configure the connection weight parameters. This can be set in the Parameters window.



Picture 6. Setting probabilities of neural connections mutations.

Under Genetics → f0, we set the probabilities of mutations on several aspects of the creature. We set all probabilities of mutation of parts, joints and neurons to zero. We only enable mutations to the connection weights (as shown on Picture 6). Also turn off cross-over in Parameters → Selection, only leaving a chance of mutation and some unchanged creatures.

Choose a fitness criterion, e.g. distance, and start the fitness-optimising evolution. It is important to choose the proper experiment definition (e.g. standard.expdef) and properly adjust all the parameters of the evolutionary process. Usually, default values can be used, but the evolutionary know-how is helpful here. After a while, the creatures will exhibit the old smooth chemotaxis.

6. **Defining new neurons.** Our new creature, however, is not much different from the old foodfinder. If we test it in the deathmatch tournament against a population of foodfinders, it won't be much better, or even worst! It will certainly make the same old mistake as the foodfinder: it wastes valuable time and energy to colliding with team members, who can offer nothing but energy loss and a probable death. Why is this?

   Well, the smell neurons, as mentioned before, detect all sources of energy – all food items and creatures (of the other teams and the own team). It would be a good idea to have smell neurons that detect all energy sources, *except* that of our team members. This can be done by defining a new neuron. Or rather, simply edit an existing example. This sounds much more difficult than it is. Neurons can be defined using *FramScript*.

   Let's edit the smell_food.neuro that you can find in the scripts folder. Open this file in an editor.

```
class:
name:Sf
longname:Smell food
description:~
Detects only food, not other creatures (in experiments with food in group #1)~
prefoutput:1
preflocation:1
vhints:32
# 32=receptor class
vectordata:~
64,5,7,100,50,90,50…etc…~
code:~
function init() {}
function go()
{
Neuro.state = LiveLibrary.getGroup(1).senseCreaturesProperty(
       Neuro.position_x,Neuro.position_y,Neuro.position_z,"energy",Neuro.creature);
}
```

~

We can edit this code, even without detailed knowledge of *FramScript*, to change the bold part in such a manner that it does not only sense creatures properties in group 1, but in all groups, except its own team. This would look like this:

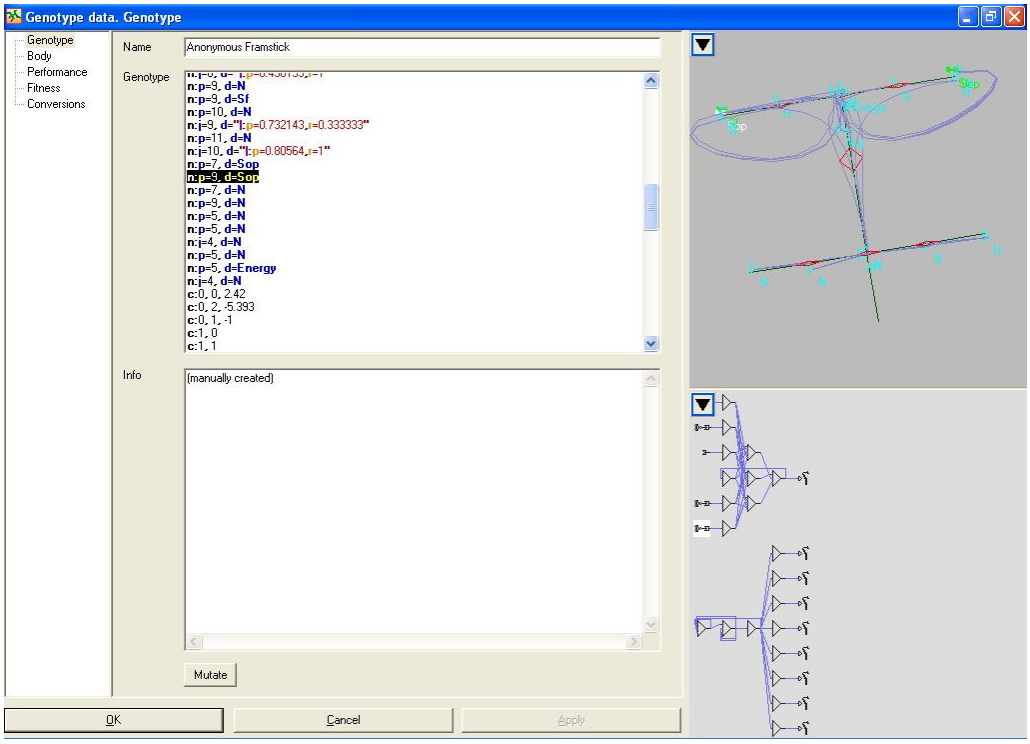
```
class:
name:So
longname:Smell other populations
description: ~
This receptor neuron detects everything except the creatures in its own population.~
prefoutput:1
preflocation:1
vhints:32
# 32=receptor class
vectordata:~
64,5,7,100,50,90,50,…etc… ~
code:~
function init() {}
function go()
{
var s=0.0;
var i=0;
while(i < LiveLibrary.groupcount)
{
        if(i != Neuro.creature.group.index)
        {
        s = s + LiveLibrary.getGroup(i).senseCreaturesProperty(
        Neuro.position_x,Neuro.position_y,Neuro.position_z,"energy",Neuro.creature);
        }
        i++;
}
Neuro.state = s;
}
~
```

Save this code under smell_others.neuro. Your new neuron will be available when you restart Framsticks. In the same way, we could make a new sensor that only smells the own population, or only food. The last seems to be a wise choise for the Deathmatch. Then we could evolve a creature that can influence its chemotaxis (moving towards other creatures, or towards food) depending on its own energy level (given by the energy level detector we added in step 5).

Making a 'smell food' neuron for the Deathmatch would be as simple as changing the getGroup(1) into getGroup(6) in the original smell_food.neuro script, because we already know that food items are in population 6.

And if we have these specialised neurons to detect energy from food items in group 6, we can re-edit the neuron above to exclude group 6. Then we have one type of neurons specialised for food, and one type specialised for creatures in other-than-own population.

7. **Using new neurons.** Once we have defined these neurons and have them available in Framsticks, we should use them in our creature. We will now have many inputs: two new smell_food neurons, two new smell_others neurons, and one energy level detector. This is a bit much for a simple perceptron to handle, since a perceptron can only discriminate linearly. That is why we must reconstruct the brain part responsible for chemotaxis to include the new neurons, and connect them to a hidden layer, which outputs to a neuron that connects to the bend muscle. We can use Fred for this, but we can also do this by hand in the Framsticks genotype editor. Adding neurons and connections is very easy.



Picture 7. New brain structure designed for more sensors.

Be sure you give the neurons a place on the body. It is especially important to give the new receptor neurons a useful place on the body (the Framsticks visualization will be very helpful here, highlighting corresponding genes, neurons in the brain, and their locations in the body). Our new creature could have a new brain like the one shown in the Picture 7. The receptors are all connected to the input layer of the neural network, the input layer is fully connected to the hidden layer, which is connected to a single output neuron. (Note that there is also a recurrent connection. Is this useful?)

Again, the performance of our creature will not be better when only defining the neural network. In the case of the perceptron, it could be possible to think about setting the connection weights (those chosen to be evolved in step 5) ourselves. In the case of this neural network, it is almost impossible. We should therefore again reconfigure the connection weights (as in step 5). However, we cannot simply use the standard experiment definition anymore!

The standard experiment definition has only two populations: one population for our creatures, and one for food items. However, our new smell_food neurons are made to detect energy sources in population #6! Evolution in the standard exerpiment definition will thus not configure the new network to anything relevant (and the neuron definition will be invalid, as it references the non-existing population #6). We have to define our own experiment definition to enable the configuration of the neural network and all the neuron types we defined. (To overcome the particular problem with food in population #2 instead of #6, we could just change the population number in our neuron to fit the standard.expdef idea, and after weight optimisation, change it back to 6).

8. **Editing experiment definition.** The standard experiment definition is a good starting point. We will have to change it to initialize 6 populations, instead of only 2. It does not matter whether the populations 3, 4 and 5 are empty or filled (as long as there are some creatures in an other-than-ours population), so we better leave them empty, to simplify the process.

   The populations are created in the onExpLoad() function:

```
                    ...
function onExpDefLoad()[8]
{
```

[8] The header of the function may look a bit different “onExpDefLoad:” in older implementations.

```
// define genotype and creature groups
if (GenotypeLibrary.groupcount != 1) GenotypeLibrary.clear();
GenotypeGroup.name="Genotypes";
if (LiveLibrary.groupcount != 1) LiveLibrary.clear();
update_fitformula();
CreaturesGroup.name="Creatures";
CreaturesGroup.nnsim=1;
CreaturesGroup.enableperf=1;
CreaturesGroup.colmask=13; //13=1+4+8
LiveLibrary.addGroup("Food");
CreaturesGroup.colmask=148; //148=4+16+128
CreaturesGroup.nnsim=0;
CreaturesGroup.enableperf=0;
              …
```

We can use this code to redefine this function to create more populations, by some copy/pasting:

```
function onExpDefLoad()
{
// define genotype and creature groups
if (GenotypeLibrary.groupcount != 1) GenotypeLibrary.clear();
GenotypeGroup.name="Genotypes";
if (LiveLibrary.groupcount != 1) LiveLibrary.clear();
update_fitformula();
CreaturesGroup.name="Creatures";
CreaturesGroup.nnsim=1;
CreaturesGroup.enableperf=1;
CreaturesGroup.colmask=13; //13=1+4+8
LiveLibrary.addGroup("Others");
CreaturesGroup.nnsim=1;
CreaturesGroup.enableperf=1;
CreaturesGroup.colmask=13; //13=1+4+8
LiveLibrary.addGroup("Empty");
CreaturesGroup.nnsim=1;
CreaturesGroup.enableperf=1;
CreaturesGroup.colmask=13; //13=1+4+8
LiveLibrary.addGroup("Empty");
CreaturesGroup.nnsim=1;
CreaturesGroup.enableperf=1;
CreaturesGroup.colmask=13; //13=1+4+8
LiveLibrary.addGroup("Empty");
CreaturesGroup.nnsim=1;
CreaturesGroup.enableperf=1;
CreaturesGroup.colmask=13; //13=1+4+8
LiveLibrary.addGroup("Food");
CreaturesGroup.colmask=148; //148=4+16+128
CreaturesGroup.nnsim=0;
CreaturesGroup.enableperf=0;
```

The food population is now population #6, as we want it. Save these changes in a file with the extension *.expdef. It will be available after a restart of Framsticks.

However, these changes will only cause the creation of 6 populations on applying or reloading the experiment definition (try it!). Nothing else it changed. We must also make sure that (1) food items are created in population 6, (2) there are other creatures in population 2.

The creation of food in the world is done by the function addfood():

```
function addfood()
{
LiveLibrary.group=1;          // change 1 into 6
if (ExpParams.foodgen=="") LiveLibrary.createFromString("//0\nm:Vstyle=food\np:");
else LiveLibrary.createFromString(ExpParams.foodgen);
}
```

Only changing the bold 1 into a 6 will cause food items to be spawned in population 6. It is as simple as that!

However, we are not done yet. Now we must edit the expdef in such a way, that the population 2 is filled with moving creatures. We take a look at the onStep() function which is called every step, and controls the creation of individuals and food:

```
function onStep()
{
LiveLibrary.group=0; // creatures
if (CreaturesGroup.creaturecount<ExpParams.MaxCreated)
{
    selectGenotype();
    if (Genotype.isValid) LiveLibrary.createFromGenotype();
}

if (ExpParams.aging>0)
{
   var i=0;
   while(i < CreaturesGroup.creaturecount)
   {
      LiveLibrary.creature=i;
      Creature.idleen=ExpParams.e_meta*Creature.numjoints
                      *Math.exp((0.6931471806*Creature.lifespan)/ExpParams.aging);
      i++;
   }
}

LiveLibrary.group=1; //food is now in livelibrary #5
                     //which is population #6, starting counting at 0
if (CreaturesGroup.creaturecount<ExpParams.feed)
   addfood();
}
```

We have to change two things here. First, copy and paste the first part (indicated by the bold function calls), and edit it such that it creates individuals in population 2. Second, we make it check whether population 6 has the specified number of food items (and not population 2, which would make it continuously create food items, since population 2 never contains any food items).

It may look like this:

```
function onStep()
{
LiveLibrary.group=0;  // creatures
if (CreaturesGroup.creaturecount<ExpParams.MaxCreated)
{
   selectGenotype();
   if (Genotype.isValid)
       LiveLibrary.createFromGenotype();
}

LiveLibrary.group=1;  // other creatures
if (CreaturesGroup.creaturecount<ExpParams.MaxCreated)
{
   selectGenotype();
   if (Genotype.isValid) LiveLibrary.createFromGenotype();
}

if (ExpParams.aging>0)
{
   var i=0;
   while(i < CreaturesGroup.creaturecount)
   {
      LiveLibrary.creature=i;
      Creature.idleen=ExpParams.e_meta*Creature.numjoints
                      *Math.exp((0.6931471806*Creature.lifespan)/ExpParams.aging);
      i++;
   }
}

LiveLibrary.group=5; // food
if (CreaturesGroup.creaturecount<ExpParams.feed)
   addfood();
}
```

We have now finished editing the experiment definition (although we could also change many more events and settings, e.g. the energy scheme). Save the expdef in the scripts folder. Load or reload this experiment definition by clicking the reload button (and check for compiler errors). Load the genotype file containing our creature with the neural network we designed, and run the experiment to see whether it works alright (i.e. whether the expdef spawns the right creatures in the right populations).

Now we are ready to re-configure the connection weights to configure the new neural network containing our own neurons (like we did in step 5). You can choose how to set the fitness criteria. Only having 'distance' won't help you much now. You will need to set the genetics parameters to only mutation, and only mutate connection weights. Set the simulated creatures to 20 or so (this will impact both population 1 and 2). And you will need to set more parameters (world settings, energy scheme etc.).

Let evolution optimise the connection weights in your creature. This could take a while. During this process, you can adjust the selection mechanism when necessary, and you may want to tweak the fitness criteria.

9. **Test in deathmatch.** After the neural network is configured to your satisfaction, you can try it in the Deathmatch experiment itself. Test it against some of your old creatures, for example. Or against the original foodfinder. Your creature should be able to beat the original every match by now. You may also try to test it against totally different creatures to see whether your one can handle them.

   After testing, return to optimising connection weights (step 5) again, or proceed by submitting your creature.

10. **Submitting your creature.** If you are satisfied with the performance of your creatures, you can decide to submit it. You only need to submit the genotype of your creature! Choose your best creature, and delete all other genotypes in the genepool. After this, save this genotype in a *.gen file (this file should contain exactly one genotype). Submit this genotype file.

7 Links

Framsticks Website:	www.frams.alife.pl
Framsticks Experimentation Center:	www.alife.pl/fec/www.index.php
Framsticks Editor	www.frams.alife.pl/dev/index.html
Virtual Life lab:	www.aisland.org/vll
Deathmatch website:	www.aisland.org/vll/projects/deathmatch.htm