Antarctica

Exploring the MAXSON Architecture

By **Luis Ángel Larios Cárdenas**

University of California
Los Angeles

UNIVERSITY OF CALIFORNIA

Los Angeles

Henry Samueli School of Engineering and Applied Sciences

Computer Science Department

Master of Science in Computer Science

Comprehensive Examination Report

Antarctica

Exploring the MAXSON Architecture

By Luis Ángel Larios Cárdenas

Student ID: 003870473

Advisor: Prof. Michael G. Dyer

Los Angeles, California, March, 2012

Antarctica

Exploring the MAXSON Architecture

INDEX

	Content	Page
	ABSTRACT	1
1.	INTRODUCTION	1
	1.1. The MAXSON Architecture	1
2.	PROBLEM DESCRIPTION AND GOALS	2
3.	THE ENVIRONMENT	3
4.	ANIMAT ARCHITECTURE	4
	4.1. Brain Circuitry	4
	4.2. Learning Using the MAXSON Architecture	6
5.	METHODOLOGY	7
	5.1. Survival Task	7
	5.2. Reproduction Task	7
6.	EXPERIMENTS AND RESULTS	9
	6.1. Project Stages	9
	6.2. Implementation Details	9
	6.3. Experiments	12
7.	CONCLUSION	21
8.	PREVIOUS WORK	21
9.	ACKNOWLEDGMENT	21
10.	REFERENCES	21
	APPENDIX	A1
	A.1. Male and Female Brains	A1
	A2. Animat Class	A4
	A3. Brain Class	A16
	A4. Simulate Function	A20

Antarctica

Exploring the MAXSON Architecture

Luis Ángel Larios Cárdenas
langel@cs.ucla.edu
University of California, Los Angeles

Abstract

The Antarctica ecosystem consists of a simulation of artificial creatures built upon the **MAXSON** neural architecture. They carry out a reproduction task that involves males and females in the emulation of *nuptial feeding* and *male brooding*, as reported in nature. Animats resort to reinforcement learning, which enables the construction of associations between objects and the satisfaction of their internal goals. The agents continuously improve their behavior, so that they effectively learn to procreate and survive by eating food and avoiding poison. Results demonstrate that the MAXSON architecture can be extended to simulate natural phenomena, with the objective of modeling biological ecosystems and the interactions within their organic and inorganic components.

1. Introduction

Animats are artificial representations of living creatures designed to operate in a tight, closed-loop interaction with their environment [Sutton 1991]. The agents are sometimes preprogrammed, so that they behave and perform tasks repeatedly. When preprograming is not possible, they are to learn which actions to take in response to their *internal goals* and external state readings.

Learning, from the animat's point of view, consists of building a mapping from situations to actions [Sutton 1998] by means of trial and error. After each action, the environment provides a numerical feedback signal, a *reward*, that the agent uses to improve its *policy* [Sutton 1991]. The objective of the intelligent agent is to make decisions that yield more positive rewards over the long term in an uncertain environment, and adjust its policy to avoid triggering actions that have a negative impact on its internal goals.

There are several *reinforcement learning* techniques for approaching the intelligent agent problem and the uncertainty related to long-term rewarding. Among them, temporal-difference algorithms [Sutton 1991] take into account the delayed effects of an action in a return-function predictor that defines which decision to make at every state.

Temporal-difference learning, however, requires a large number of iterations for modeling even simple behaviors [Crabbe 2000a]. Such a restrictive training time makes temporal-difference learning unsuitable for animat development in virtual environments; the agent should get enough experience from interacting a minimum number of times with objects, and continue reinforcing its positive/negative associations as it perceives the same objects without a direct trial.

In order to define an animat-oriented reinforcement learning framework, [Crabbe 2001] introduced the MAXSON architecture, which is a flexible neural model that learns faster than traditional temporal-difference approaches while balancing the requirements of multiple simultaneous goals.

1.1. The MAXSON Architecture

The MAXSON architecture (figure 1) contains two sub-networks: a second-order *policy network* and a first-order *value network* [Crabbe 2001]. The policy network dynamically generates the action that the animat has to take at every time step, while the value network yields external reinforcement signals in response to the agent's effect in the environment.

In order to behave intelligently, an animat has to take actions according to its goals [Crabbe 2001], such as those related to creature's survival and

other more sophisticated behavior. Goals that fluctuate over time may eventually come into a conflict. The agent is able to solve conflicting goals through the second-order connections [Beer 1990] in its policy network. Second-order connections multiply pairs of input neurons before their signals are summed. Second-order connections serve as gating (multiplicative) connections that modify the strength of node-to-node, first-order connections between input and output neurons (figure 1) [Werner 1994].

The animat receives external reinforcement as it performs actions and satisfies or fails in its goals. However, these rewards are intermittent signals that require multiple goal satisfactions for learning to happen. The value network allows the animat to learn from a single interaction with the world by transforming intermittent rewards into immediate reinforcement that the policy network can use for learning at every time step [Crabbe 2001].

The policy network is made up of sensor and output neurons. Sensor nodes are separated into two groups: the *external input neurons* (I_i), and the *internal goal nodes* (G_g). The external inputs are triggered in response to the animat sensing objects in its surroundings. The goal nodes are inputs coming from within the agent's body and define its current internal state [Crabbe 2001]. When the agent detects the activation of its inputs, their signals are multiplied via second-order connections with weights ($W_{i,g,o}$), summed up, and fed into *output* neurons (O_o) that correspond to the agent's actions.

Learning in the policy network consists of adjusting the weights of its second-order connections using *immediate, distributed* reinforcement [Crabbe 2001]. Each time step, the reinforcement signals (R_i) are computed as a function of the corresponding input sensor reading with respect to the previous time step. This results in tuning the creature's policy whenever the agent's external inputs decrease or increase as the animat moves away or approaches objects related to the satisfaction or failure in its internal goals.

The value network consists of connections from the goal sensors to the external input nodes [Crabbe 2001]. It re-calculates the reinforcement signals (R_i) for the external sensors that the policy network uses at each time step.

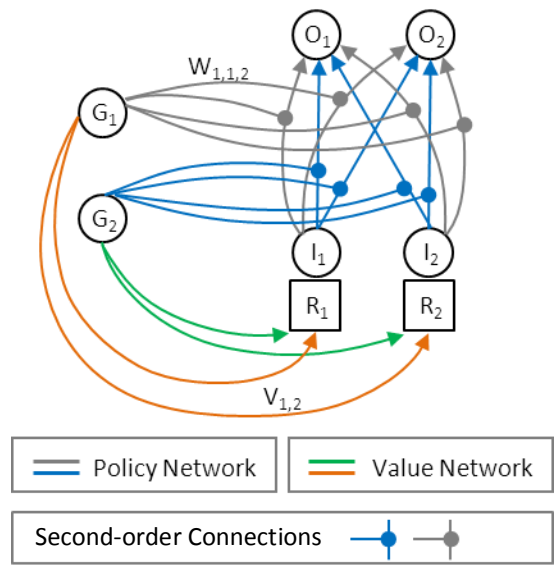


Figure 1. Simple model of the MAXSON architecture.

Learning in the value network involves adjusting its weights ($V_{g,i}$) when a change in a goal sensor indicates that the animat has satisfied or failed in an internal goal [Crabbe 2001].

The MAXSON architecture allows agents to learn about their environment quickly and accurately. Animats learn from a single interaction with objects by using their value network, and achieve multiple simultaneous goals through the second-order connections in their policy network. Consequently, agents that integrate MAXSON-based brains take advantage of the environment they inhabit and maximize their chances of success, in contrast to implementing other reinforcement learning frameworks, such as the temporal-difference approaches [Crabbe 2001].

2. Problem Description and Goals

Reinforcement learning and animats are the core components of the Antarctica ecosystem: an artificial environment that introduces the MAXSON architecture to the simulation of two animal behaviors: *nuptial feeding* and *male brooding*.

Nuptial feeding is the act of bringing food to a female before and/or during copulation. The act has varying purposes for different species. For

instance, some insect males like the dance fly (*Empis Borealis*) capture prey before the intercourse as a means of distraction. The male presents this prey as a *nuptial gift*, keeping the female still as she consumes the prey during copulation, thus allowing the male enough time to deposit his sperm [Svensson 1987]. In an example from the bird kingdom, the Northern cardinal (*Cardinalis Cardinalis*) male presents a morsel of food to his mate during courtship. However, in contrast to the dance fly, he will continue to nourish her throughout the entire incubation period if mating succeeds [Elliot 1998].

Male brooding is a reversal of parental sex roles in which males incubate eggs after mating with females. The Emperor penguin (*Aptenodytes Fosteri*) is an exemplar of this behavior. A male Emperor penguin must take care of the egg while the female forages to replenish her depleted energy reservoir [Williams 1995]. This incubation period is a major commitment that can last up to 64 consecutive days of fasting [Mrosovsky 1980] while taking special care of an offspring that could easily die if it touched the frozen ground.

The Antarctica ecosystem animat carries out a reproduction task, resembling documented nuptial feeding and male brooding in birds, while it learns how to survive the harsh conditions of the environment via reinforcement learning, using the MAXSON architecture.

At the start of the simulation, animats know nothing about the dynamics of their world. They are only susceptible to feeling "relief" or "pain" as a consequence of undertaken actions. These rewards or punishments provide them with feedback for creating associations between external situations and the satisfaction or failure in their internal goals.

3. The Environment

The virtual world consists of a two dimensional, continuous space enclosed by 4 walls. The space is subdivided into a 20 x 20 grid, with each cell[1] (or *bin*) varying in temperature: "warm" or "cold." Periodically - every 2,000 or 5,000 time steps, the

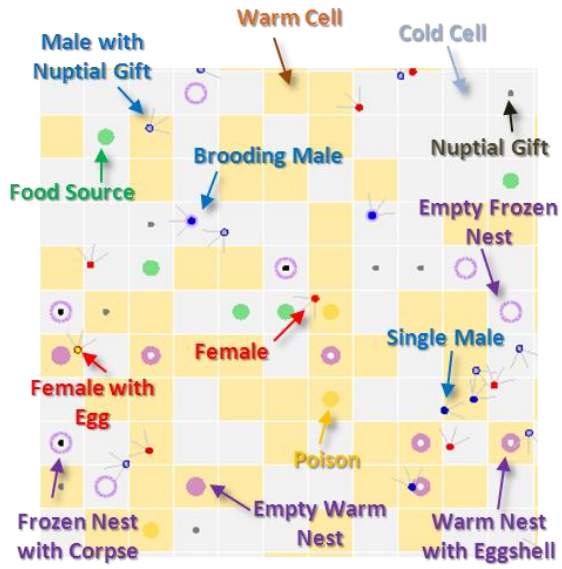


Figure 2. The environment, the inert objects, and the living creatures.

cells' temperature shifts, simulating a flux of heat in the ecosystem.

There are two classes of *entities* inhabiting the Antarctica ecosystem: *inert objects* and *living creatures* (figure 2).

Inert objects are represented by immobile, circular entities scattered randomly throughout the terrain. There are five types of inert units:

a) **Food sources** are non-perishable energy providers that feed animats on contact. They bloom in cold cells but become poison when temperature shift changes the underlying bin from cold to warm.
b) **Poison** units are food sources that bloom in warm areas. They cause a generalized "pain" to animats that touch them. Poison units become food sources when their underlying cell's temperature goes from warm to cold.
c) **Warm nests** are objects where males leave eggs for hatching. They are located in warm cells but become *frozen nests* when the underlying cell's temperature changes from warm to cold.
d) **Frozen nests** are nests located in cold bins. Eggs that males leave on these objects die immediately. They become warm nests when a temperature shifting changes their cells from cold to warm.
e) **Nuptial gifts** are pieces of food (distinct to food sources) that males pick up for

[1] In the Antarctica ecosystem, 1 distance unit = length of a cell.

presenting to a female during the mating process.

The living creatures are animats that explore the environment and interact with or perform actions on other objects. In Antarctica, agents' *interaction* or *contact* is defined as the act of being aligned at a distance of 0 to 3/8 units from the target's position. The interaction angle is computed according to the type of item:

a) Agents interact with *inert objects* at [-60º, 60º].
b) Agents interact with other *living creatures* at [-90º, 90º].

Agents and inert objects have a collision radius equivalent to 1/8 units (figure 3). As animats navigate the environment, they perform collision detection and avoidance [Buckland 2005] with objects that intersect a bounding box extending ¼ units in front of them (known as *facing* direction).

When an agent makes a decision or computes a new direction, it transforms its *velocity* into a *steering force* [Buckland 2005] [Hill 2001] that, combined with the accumulated obstacle and wall avoidance forces, determines the final velocity and facing direction.

4. Animat Architecture

The animats are massless particles that sense their environment in a 180-degree *field of view*. The range of view extends a 1-unit radius in their facing direction. They emanate a *temperature* that reaches a 3/8-unit radius. The agent's temperature allows animats to sense one another beyond the 1/8-unit personal (collision) radius.

Creatures are assigned the following ***features*** when they are spawned:

a) ***Energy capacity***, which ranges from 6,700 to 10,000. It is equivalent to the number of time steps the animat can survive without feeding.
b) ***Reproductive completeness capacity***, which defines the animat's "apathy" for reproducing. It has the same value as the *energy capacity*.
c) ***Parenting completeness capacity***, which defines the animat's "tranquility" with respect to leaving an egg in a nest for hatching. Its value matches the *energy capacity*.

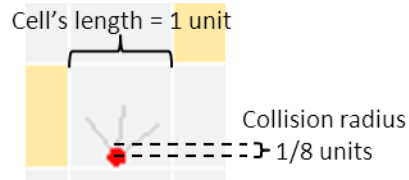


Figure 3. The collision radius with respect to the unit of distance.

Throughout their lives, agents update their *energy*, *reproductive completeness*, and *parenting completeness* ***feature levels*** in response to the actions they perform in the environment. The difference between the current levels and their corresponding capacities generates ***state indicators***, also referred to as ***internal goals***. The agent's purpose is to determine what actions to take in order satisfy its goals. Animats satisfy goals when the following *state indicators* drop to zero:

a) **Hunger** is the difference between *energy capacity* and *energy level*.
b) **Libido** is the difference between *reproductive completeness capacity* and *reproductive completeness level*.
c) **Brood** is the difference between *parenting completeness capacity* and current *parenting completeness level*.

Animats are born as one of two sexes: *male* or *female*. Their gender defines which feature levels they adjust in accordance to their role in the reproduction task:

a) Females vary only their *energy level*.
b) Males update all 3 *feature levels*.

This difference in how males and females adjust their *feature levels* allows sexual differentiation in the same brain circuitry.

4.1. Brain Circuitry

The MAXSON architecture depicted in figure 4 illustrates the animat's brain. It is made up of a collection of nodes that are grouped in various categories:

a) **External input nodes** activate in a range of [0, 1] in response to perceiving entities in the environment. Some inputs trigger in both sexes, while others just in females or just in males:
   - **Food Left/Right** activates when the animat sees food sources.

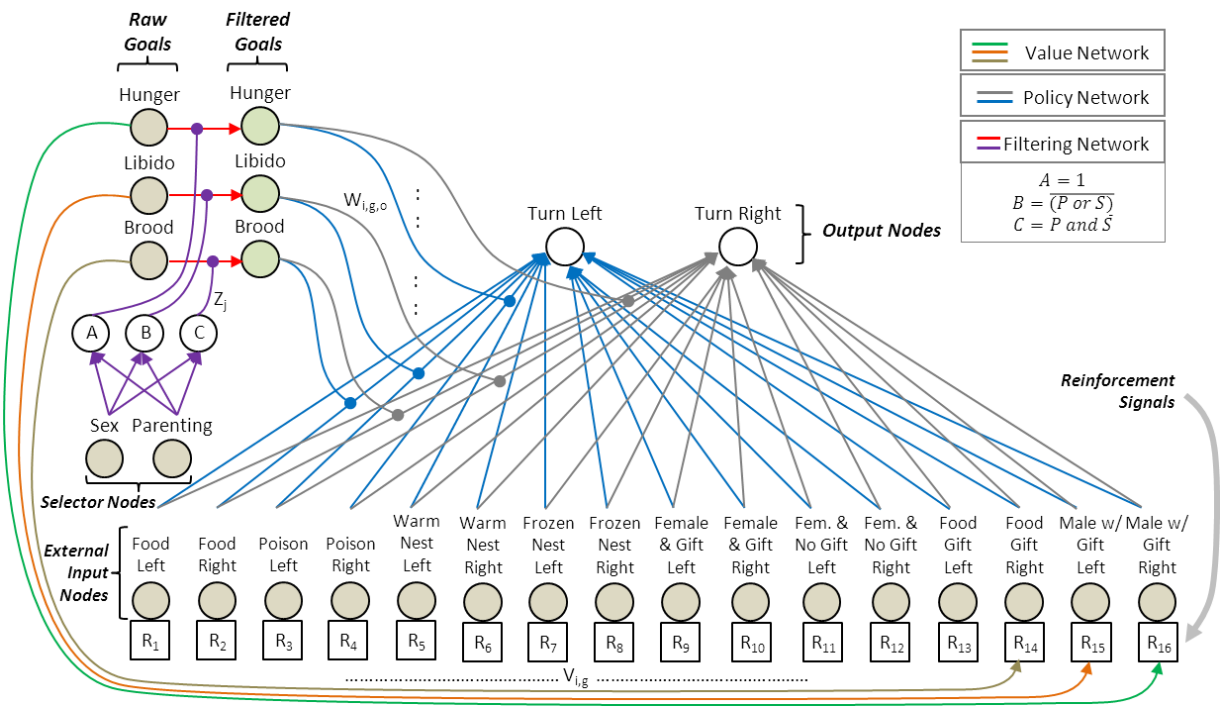


Figure 4. Animat's brain using the MAXSON architecture.

- **Poison Left/Right** activates when the animat sees poison units.
- **Warm Nest Left/Right** activates when the male agent sees a nest in a warm cell.
- **Frozen Nest Left/Right** triggers if the male agent detects a nest in a cold cell.
- **Female & Gift Left/Right** activates if the male senses a female with an egg while he is carrying a nuptial gift.
- **Female & No Gift Left/Right** triggers if the male perceives a female with an egg, but he is not carrying a nuptial gift.
- **Food Gift Left/Right** activates when the male detects nuptial gifts in the terrain that he can pick up.
- **Male with Gift Left/Right** allows females to sense males that have a nuptial gift with them.

b) **Output nodes** produce values from 0 to 1 when they activate. These values are scaled to represent 0 to 4 degrees of turn between the object and the current facing direction. *Turn left* and *turn right* trigger in parallel, resulting in the agent following the direction of the larger one.

c) **Raw-goal nodes** are equivalent to the *internal state indicators*. Their activation feeds into the value network to compute external reinforcement weights:

- **Hunger** represents the percentage of the current *hunger state indicator* with respect to *energy capacity*.
- **Libido** represents the percentage of the current *libido state indicator* with respect to the *reproductive completeness capacity*.
- **Brood** represents the percentage of the current *brood state indicator* with respect to the *parenting completeness capacity* value.

d) **Filtered-goals** are raw-goal nodes whose signals are enabled by the *selector nodes* activation:

- **Hunger** is activated directly from the *hunger raw-goal* in both males and females.
- **Libido** is activated from the *libido raw-goal* only in males whose *parenting* selector node is not triggered (i.e. single males).
- **Brood** gets the activation from the *brood* raw-goal, only in males whose *parenting* selector is triggered (i.e. brooding males).

e) **Reinforcement nodes** provide immediate internal reinforcement for each external input. These signals are generated in response to the perception of objects in the animat's field of view.

$\vec{G}$ is the raw-goal sensor input vector at current time step.
$\overrightarrow{G^{t-1}}$ is the raw-goal sensor input vector at the previous time step.
$\vec{V}$ is the 2D weight matrix of the value network.
$\vec{I^t}$ is the external input sensor vector at the current time step.

$i = \arg\max_i \vec{I_i^t}$
For each g in $\vec{G}$ do
$\overrightarrow{V_{g,i}} = \overrightarrow{V_{g,i}} + \vec{I_i^t} \times \delta_2$ where

$$\delta_2 = \begin{cases} \overrightarrow{G_g^{t-1}} - \overrightarrow{G_g^t}, & if \ \left|\overrightarrow{G_g^{t-1}} - \overrightarrow{G_g^t}\right| > \theta_2 \\ 0, & otherwise \end{cases}$$

Figure 5. Algorithm for adjusting the first-order value network weights.

C is a scalar constant.

$g = \arg\max_g \vec{G}$
For each i in $\vec{I^t}$ do
$\vec{R_i} = \left(\vec{I_i^t} - \overrightarrow{I_i^{t-1}}\right) \times \overrightarrow{V_{g,i}} \times C$

Figure 6. Algorithm for computing immediate internal reinforcement at each external input sensor node.

f) **Selector nodes** enable and disable behaviors depending on animat sex and parenting state:
   - Females (**sex** = 1) only activate their *hunger* raw- and filtered-goal nodes.
   - Males (**sex** = 0) go from singleness (**parenting** = 0) to brooding (**parenting** = 1).

The MAXSON-based brain contains a **filtering network**, in addition to the traditional policy and value sub-networks. The filtering network triggers animat behavior when it allows raw-goal signals to pass to the filtered-goal nodes. It inputs the selector node values to Boolean-functioned neurons whose outputs are then used to enable filtered-goal activation via second-order connections.

The objective of the filtering network is to isolate external rewards that affect raw-goal nodes, from the filtered-goal values that the animat feeds into its policy structure. The filtering network then allows the integration of ***delayed reinforcement*** and ***sexual differentiation*** in the same architecture. It permits the adjustment of the value network weights from external reinforcement at the current time step, but it uses those associations to update the policy network until the animat enters another life stage (e.g. when the *parenting* selector node is 0, males update their value network, but the policy network uses such reinforcement to learn brooding behavior when the *parenting* selector node is 1).

4.2. Learning Using the MAXSON Architecture

Both males and females are born with a MAXSON-based brain whose weights are initialized to 0 in the value sub-network, and to a random value in the range of [0.494, 0.506] in the policy network.

Zero-valued weights in the value network correspond to having no positive or negative associations with entities. The animat's objective is to build these associations from the rewards that the environment provides when the agent interacts with objects and satisfy/fails in its goals[2]. The algorithm that describes learning in the value network is depicted in figure 5.

The external reinforcement registered in the connections between raw-goal nodes and external input neurons in the value network is used to generate internal, immediate rewards. The internal reinforcement signals are calculated every time step, following the process described in the algorithm of figure 6.

Finally, the algorithm in figure 7 shows how the internal reinforcement signals are used for adjusting the weights in the second-order connections of the policy network, as the agent perceives a change in its external input node values. In the long term, the animat learns to approach entities that satisfy (decrease) its *hunger*, *libido*, or *brood* goals and avoid those that cause failure in (increase) them.

The learning algorithms of figures 5 and 7 use two thresholds that remove undesirable effects:

[2] We will refer to the terms "internal goal" and "raw-goal node" simply as "goal" when both represent the same concept.

W is the 3D weight matrix.
$\vec{O}$ is the network output vector.
$\vec{R}$ is the reinforcement vector.
$\overrightarrow{I^{t-1}}$ is the external input sensor vector at the last time step.
$\vec{G}$ is the filtered-goal sensor input vector at current time step.

$o = \arg\max_o \vec{O}$
$g = \arg\max_g \vec{G}$
For each i in $\overrightarrow{I^t}$ do

$$w_{i,g,o} = w_{i,g,o} + \overrightarrow{R_i} \times \overrightarrow{O_o} \times \delta_1 \text{ where } \delta_1 = \begin{cases} 1, & if \ \left|\overrightarrow{I_i^{t-1}} - \overrightarrow{I_i^t}\right| < \theta_1 \\ 0, & otherwise \end{cases}$$

Figure 7. Algorithm for adjusting policy network weights.

a) θ_1 accounts for the presence of objects that suddenly appear in the field of view and could cause a disproportional change in a policy's second-order connection weight. Its current value is 0.022222, as it is in [Crabbe 2001].
b) θ_2 allows distinguishing between actions that satisfy/cause failure in goals and the natural metabolism[3] of the animat. Its current value is 0.02, following the guidelines in [Crabbe 2001].

5. Methodology

In the Antarctica ecosystem, animats must learn to survive and reproduce, simulating birds' nuptial feeding and male brooding.

5.1. Survival Task

Agents learn to survive by approaching food when their *hunger* goal is not zero, avoiding poison at all times.

Interaction with food sources, or *feeding*, replenishes the agent's *energy level* (it drops the *hunger* goal to zero). Consequently, the animat receives a positive external reward which registers an increment in the value network connection weight between the *Food* input node and the *hunger* goal.

Unfortunate agents that interact with or *consume* poison receive an external reinforcement that decreases all of the animats' *feature levels* by 2.5%. A single contact with poison registers negative rewards in the connections between the *Poison* input node and each of the goals, emulating the feeling of a "generalized pain."

The agent's *energy level* decreases at a rate of 0.01 units per time step as part of its internal metabolism. As the animat's *energy level* decreases, *hunger* increases. This natural *hunger* increment and the continuous interactions with poison reduce *energy level* towards zero. Consequently, creatures die and disappear from the environment when their *energy level* reaches zero.

5.2. Reproduction Task

The animats' ultimate purpose is reproducing in order to maintain the continuity of their species. However, males and females must learn their roles in the procreation process.

5.2.1. Female's Role

A female' role in the reproduction task consists of learning to approach males in order to replenish her *energy level* by nuptial feeding. Females are preprogrammed to reject males that interact with them without a nuptial gift; however, when both a male carrying a nuptial gift and a female come into contact, they *mate*, and she automatically gives her offspring (an egg) to him for brooding.

Females are continuously producing eggs for giving them up to males that bring a nuptial gift in exchange.

A female can perceive males and whether or not they are carrying nuptial gifts. Depending on the condition of the male that interacts with her, she performs one of the following actions:

[3] Metabolism is defined as the regular increment and decrement of feature levels at each time step.

a) She rejects the male if he does not have a nuptial gift.
b) She mates, gives her egg to the male, and eats the nuptial gift. Nuptial feeding replenishes her *energy level* and installs a positive external reinforcement in the value network connection that links her *Male with Gift* input neuron with the *hunger* goal.

Females cannot pick up nuptial gifts from the terrain for eating, though they have the same effect as feeding during *copulation*.

When the female gives up her egg to a male, she has to wait for 20*(*energy capacity*) time steps to be ready for mating again. During her resting period, she continues exploring the environment, feeding, and avoiding poison in order to survive.

5.2.2. Male's Role

Males are designed to sense and interact with females carrying eggs, nuptial gifts, frozen nests, and warm nests. Males automatically perform actions such as mating, picking up nuptial gifts, and laying eggs in any nest, depending on the entities they interact with and their current stage of life. The male's role in the reproduction task consists of:

a) Learning to approach nuptial gifts in order to pick them up.
b) Learning to approach females with eggs when he has a nuptial gift in order to satisfy his *libido* goal.
c) Learning to avoid females with eggs when he is not carrying any nuptial gift, so that he is not affected by females' rejections.
d) Learning to approach warm nests for laying eggs in order to decrease his *brood* goal.
e) Learning to avoid interacting with frozen nests when he is carrying an egg, so that his offspring will not die.

Males have two life stages according to their selector nodes in their MAXSON-based brain: *singleness* and *brooding*. These stages happen before and after mating with a female respectively.

Males' *parenting* selector node is disabled during singleness. At this stage, both the *hunger* and *libido* goals (that pass as is through the filtering network) determine the animat's behavior. Male's *libido* internal goal is a cyclical signal that metabolically increases at a rate of 0.008 and decreases at 0.018*(*reproductive completeness capacity*) when its value reaches 60% of its *feature* capacity.

Single males automatically *pick up* nuptial gifts when they interact with such inert objects during exploration Males are designed to carry nuptial everywhere they go, until they interact with females that have eggs. The reproduction task success depends on the possession of these items when males decide to approach females for mating.

Mating, from the male's point of view, is the interaction with a female that has an egg ready to give up. *Copulation* occurs when both male and female are in the field of view of each other, at a distance of 3/8 units. Mating provides the male with an external reward that depends on whether or not he has a nuptial gift:

a) If he has a nuptial gift, he gets the female's egg, and his *libido* goal drops to zero. The satisfaction of his *libido* generates a positive reward update in two connections of the value network:
   - Between the *Female & Gift* external input node and the *libido* goal.
   - Between the *Food Gift* external input node and the *libido* goal.
b) If he does not have a nuptial gift, the female rejects him, and he perceives a reduction of 2.5% of his *reproductive completeness level*. Failure in his *libido* registers a negative update in the connection between the *libido* goal and the *Female & No Gift* external input node in the value network.

A male's life stage is preprogrammed to automatically change from singleness to brooding by turning on the *parenting* selector node after he has received the egg from his mate. At this new stage, the *libido* filtered-goal is inhibited and he cannot reproduce any longer.

During brooding, the *hunger* and *brood* goals are active, feeding their respective portions in the policy sub-network. The task of the brooding male is to find a nest for leaving the egg he is preprogrammed to carry, while still fulfilling the survival task.

Initially, brooding males leave their eggs in any nest (frozen or warm) they interact with. At this point, they do not know which type of nest is

suitable for the egg to hatch. Males learn to identify the effects of nests from the evidence that the environment provides. There are two types of evidence:

a) **Birth evidence** are "eggshells" that newborns leave when their parents lay them in warm nests.
b) **Corpses** are the "remains" of eggs that died when their parents left them in a frozen nest.

Birth and corpse evidence stay in their respective nests until temperature shifts in the underlying cell (every 2,000 or 5,000 time steps). The process of removing evidences by changing warm nest into frozen nests (and vice versa) is known as ***evidence cleaning process***.

Males interact with evidence, regardless if they are single or brooding. Neither do they distinguish if the remains belong to their children or to other's. The interaction with evidence in nests generates external reinforcement as follows:

a) **Birth evidence** increase male's *parenting completeness level* by 2.5%. This increment positively updates the link between the *brood* goal and the *Warm Nest* input node in the value network.
b) **Corpses** decrease the *parenting completeness level* by 2.5%. This reduction negatively updates the link between the *brood* goal and the *Frozen Nest* input node in the value network.

The combination of the *brood* raw-goal and the filtering network (figure 4) allows a male to learn brooding behavior even when he is single. The delayed external reinforcement yields two advantages:

a) It removes the necessity of waiting for creating associations with nests until *brood* is the maximally driving goal.
b) Eggs have higher probability to hatch since their parents have already learned that the offspring die if they leave them in frozen nests.

Leaving the egg in a nest automatically relieves the male from being in the brooding stage. At that moment, he turns off the *parenting* selector node and *libido* is again taken into account in its policy network. Once again in his singleness stage, he will continue looking for a mate for reproducing while still complying with the survival task.

6. Experiments and Results

The following stages allowed us to develop the Antarctica ecosystem and implement the MAXSON architecture as the core of the animats' brain.

6.1. Project Stages

1. *Animats perform obstacle and wall avoidance.*
   - They move randomly throughout the terrain and interact with food sources, poison units, warm and frozen nests, nuptial gifts, and other animats.

2. *Animats learn to survive.*
   - They are provided with a MAXSON structure for a brain. Their task consists of learning the policy that allows survival by eating food and by avoiding poison.

3. *Males and females learn to reproduce.*
   - Males that have a nuptial gift learn to approach females for mating.
   - Females learn from nuptial feeding, so that later they feel attracted to males that carry a nuptial gift.
   - Males identify warm nests as the right places to leave the eggs they receive after copulating. They must learn that frozen nests kill their offspring on contact.

4. *A temperature-varying environment.*
   - The initial temperature assigned to cells shifts periodically, so that food becomes poison and warm nests turn into frozen nests.

6.2. Implementation Details

Each time step, the Antarctica ecosystem simulation loop executes in the following way:

1. The environment updates animats' metabolism.
2. The agent's policy network defines the next action to take.
3. The animat performs the action in the environment.
4. The animat senses the new environment configuration.

5. The environment provides external reinforcement in the case the agent interacts with an entity and satisfies/fails in a goal.
6. The policy network weights are updated via internal reinforcement.
7. The value network weights are adjusted in case the agent satisfied/failed in a goal.

Implementing the simulation loop brought up several details related to the physics of the environment and the intended level of accuracy.

6.2.1. Sensors Design

The initial number of nodes for each external input was 4, in the same way as in [Crabbe 2001]. The nodes were object's *angle left/right* and *distance left/right*, registering the angle and the distance to the closest target item respectively. As the animat moved towards the object, or the object had a smaller angle with respect to the agent's facing vector, the activation of the input neurons approached 1. Nonetheless, we noticed that agents with this configuration tended to learn incorrect policies, e. g. turn to the left when there was food to their right. The resulting wrong policies appeared because the angle neuron had the largest influence in determining the animat's action, disregarding the distance input node.

In order to solve the issue, we blended both angle and distance inputs nodes into one sensor. The activation of this sensor was based on the animat's facing vector and the target's position. In figure 8, for example, the current agent's facing vector (solid arrows) determines if it is approaching or moving away from the object by taking into account the facing vector in the previous time step (dotted lines).

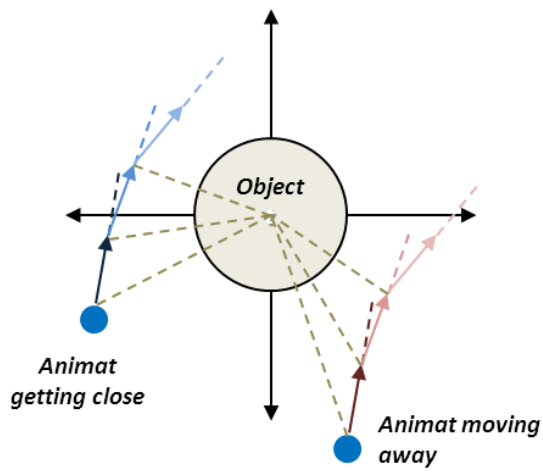


Figure 8. Difference of approaching and moving away from a target when the facing vector is in the same or opposite object's side with respect to last time-step facing.

Our formulation of the sensor node assigned a quadratic activation function ($f(x) = -x^2 + 1$) with positive ordinate values for approaching (figure 9), and negative ones for moving away ($f(x) = x^2 - 1$) (figure 10). The abscissa to these functions corresponded to the normalized distance to the object, while the angle was embedded in a change of coordinate systems operation. The reference frame transformation helped identify if the current facing vector was in the same or opposite side of the object's position (blue and red arrows in figure 8).

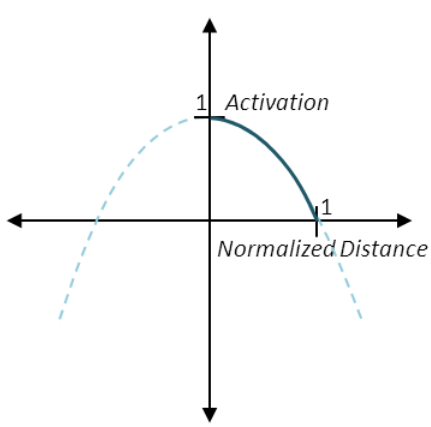


Figure 9. Activation function for approaching behavior.

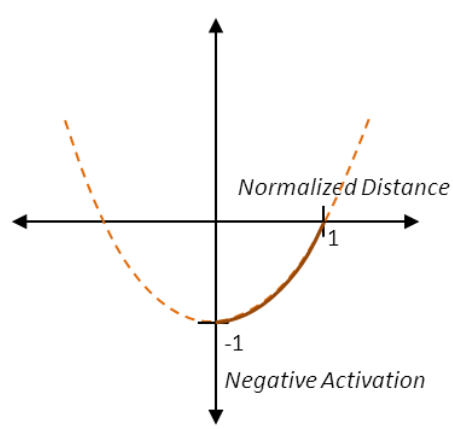


Figure 10. Activation function for moving-away behavior.

Unlike the angle-distance pair of nodes, our newly designed sensor activation decreased as the agent reduced its distance from the target and kept steering its facing vector away from the object (figure 8). The negative change in the ordinate value indicated that the agent should receive immediate reinforcement for moving away (algorithm in figure 6), rather than from approaching. Our sensor thus removed the effect of solely taking into account either distance or angle, allowing agents to correctly learn to approach positive-rewarding objects and avoid negative-rewarding units.

6.2.2. Pain Goal Node

The current implementation of Antarctica does not possess an individual goal node for registering pain like it is suggested in [Crabbe 2001]. We observed that a pain goal had no effect in the learning process because pain never took over the animat's behavior, in contrast to *hunger*, *libido*, or *brood*.

In our artificial ecosystem, experiencing pain increased the maximally responding goal (or all of them in the case of interaction with poison). Such increment allowed the agent to associate a negative reward from the interaction with an object to its current behavior. Later, the negative associations were used for adjusting the policy second-order connection weights with immediate, internal rewards when the agent continuously observed the same kind of objects during exploration.

6.2.3. Initialization of Weights in the MAXSON Architecture

The initial weights in the connections of the MAXSON structure had a profound effect in learning and in the animats' performance. We found that assigning random values to weights in broad ranges, like [0, 1] for the policy second-order connections and [-1, 1] for the value network links, prevented 40%-60% of the population from approaching food sources and avoid poison units. Agents with this brain configuration were not able to comply with the survival task and died before the end of the simulation.

We avoided the initialization problem by setting the animats' MAXSON architecture to zero-valued weights in the value network, and to random weights in the range of [0.494, 0.506] in the second-order connections of the policy network. Our initialization allowed the appearance of learning only after the agent started interacting with the environment entities, rather than before.

6.2.4. Output Nodes Activation

The output nodes in the policy network (with reference to figure 1) compute their value from the following expression:

$$O_o = \sum_{i \in I, g \in G} I_i G_g W_{i,g,o}$$

The equation implies that the magnitude of the output signals depends on the number of non-zero $I_i G_g W_{i,g,o}$ terms that contribute to their activation.

This calculation caused a difference in the performance of males and females since males had two active filtered-goal nodes at any time (*hunger* and *libido*, or *hunger* and *brood*), while females had only one (*hunger*). Consequently, males had an advantage in the survival task because they fed their output neurons with more terms than females. Thus, males' larger output activations saved them from continuous interaction with poison, whereas females showed a slower response that could not help them.

We attempted to solve the unbalanced outputs between sexes through the normalization of both *turn left* and *turn right* neurons activation. However, normalization made agents stop obeying the magnitude of their external input nodes signals, responding equally when being far and close to an object. The observed effect consisted of sharp turns when just a portion of an entity was barely visible in the animat's field of view.

We decided to continue having non-normalized outputs activation and separated experimentation data by sex. Females, despite being in disadvantage relative to males, learned the expected policy for survival; however, females' number of consumed poison objects was, on average, 1.8 to 3.3 more units than males' in experiments that involved learning.

6.2.5. Potentiating Learning

The original MAXSON learning algorithms scale reinforcement signals and weight updates by multiplying the contributing terms with goal

values, outputs activation, and input node signals. Scaling maintains the computed adjustments in their corresponding ranges of [-1, 1] in the value network, and [0, 1] in the policy network. However, in our current implementation, scaling had the effect of slowing learning since the calculated adjustments were on the order of 10^{-4} to 10^{-3}. Therefore, we removed some of the scaling factors and introduced a constant (C = 10 for algorithm in figure 6), favoring the learning speed while maintaining connection weights within their respective range.

6.2.6. Disabling Sensing Temporally

Inert objects are always available in their original cell, except nuptial gifts, which disappear when males pick them up, reappearing 500 time steps later. The immobility of inert items made agents move in circles around positive-rewarding entities and experience pain repeatedly from negative-rewarding units.

We disabled sensing objects immediately after getting an external reward in order to avoid the circling and multiple injuring effects. Thus, animats became immune to that class of rewarding entities for 3*(*energy capacity*) time steps and were able to resume the activities according to their tasks.

6.3. Experiments

The Antarctica ecosystem is built upon the hypothesis that the MAXSON architecture provides a flexible reinforcement learning framework that allows animats to improve their policy for survival and reproduction, as means of ensuring the continuity of their species.

All of the present experiments ran during 50,000 time steps on a terrain with 20 units of each type of inert object. We chose this quantity based on preliminary simulations. Such number of items allowed learning and survival of more than 90% of an initial population. Food sources, poison units, warm nests, frozen nests, and nuptial gifts were randomly distributed at individual cells. The environmental bins were assigned warm or cold temperatures with probability of 0.45 and 0.55 respectively.

We gathered experimentation data separately for males and females at each time step. In each experiment, we analyzed the ***cumulative average*** of the following parameters for a control group:

a) The number of times that males fed, tasted poison, left eggs in warm nests, and were rejected when they interacted with females.
b) The number of times that females consumed food, tasted poison, and received nuptial gifts when they mated with males.

In our ecosystem, animats' sex, hunger level, and life stage are represented via colors as it is illustrated in figure 11. The various inert objects that agents came across in exploration are described in figure 12.

6.3.1. Random-Decision-Based Animats

The first experiment evaluated the performance of 30 males and 30 females that did not learn any of the survival and reproduction tasks. The animats did not spawn agents when males left eggs in warm nests. The environment dynamically shifted the cells' temperature every 5,000 time steps, affecting the distribution of inert objects.

The objective of this experiment was to obtain a reference for comparing further results where learning helped animats satisfy their multiple goals. Figure 13 depicts the population performance from data collected in experiment 1.

Random-decision-based animats did not survive as

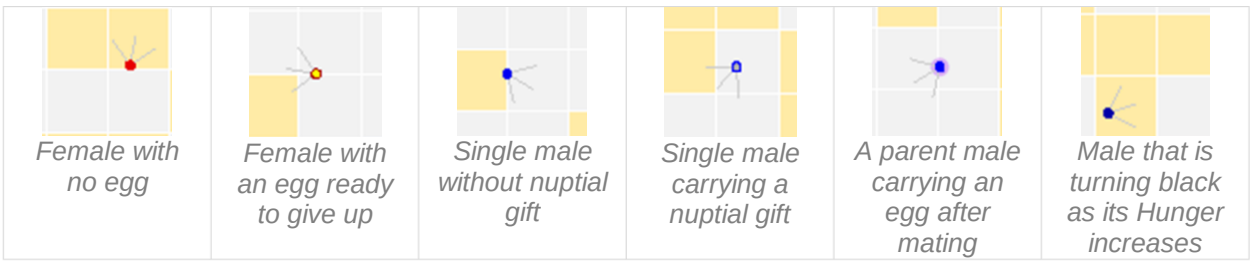


Figure 11. Visual display of animats' life stages.

Figure 12. Environmental inert objects.

a society by the end of the simulation. The control group reduced from 60 creatures to 7 at the 50,000th time step.

The agents interacted with food sources and poison units with the same proportion. However, the average started to represent the performance of only half of the population around the 23,600th time step.

Nuptial gifts and hatched eggs did not follow the feeding and poisoning trend because of animats' mobility. In general, we observed that it was more difficult for a male to find a female than an inert object.

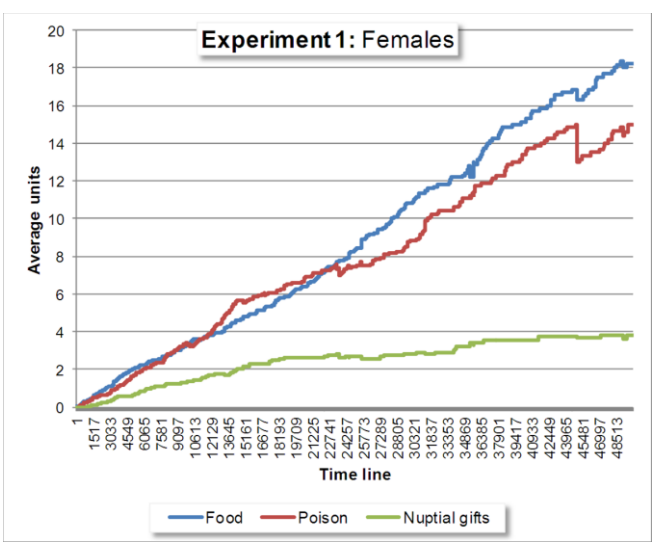
Rejections increased continuously because males did not have a way to avoid interactions with females when carrying no nuptial gift. Rejections closely followed the frequency of poison consumption at the beginning of the simulation, but started to decrease with the reduction of the population

6.3.2. Learning with Various Field-of-View Radii

Experiments 2, 3, and 4 enabled learning but with different agents' field-of-view radii: 1, 1.5, and 2 units respectively. 30 males and 30 females explored the environment using inverted parabolic activation in their MAXSON external input nodes (figures 9 and 10). However, the creatures were not allowed to spawn new agents when males left eggs in warm nests.

Half of the females were born with an egg ready to give up, and half of frozen and warm nests began with their respective type of evidence. The environment dynamically shifted the temperature in cells every 5,000 time steps. Birth and corpse evidence in nests disappeared during the *cleaning process*, in which underlying bins went from warm cells to cold cells and vice versa.

Figures 14, 15, and 16 plot the performances of animats when their field-of-view radii were 1, 1.5, and 2 units respectively. In experiment 2, 27 females and all of the males survived by the end of the simulation; in experiment 3, 28 females and 27



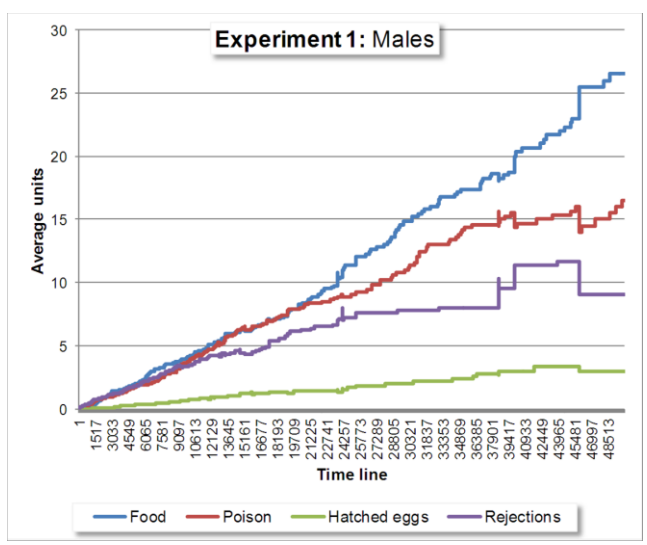


Figure 13. Experiment 1: Random-decision makers.

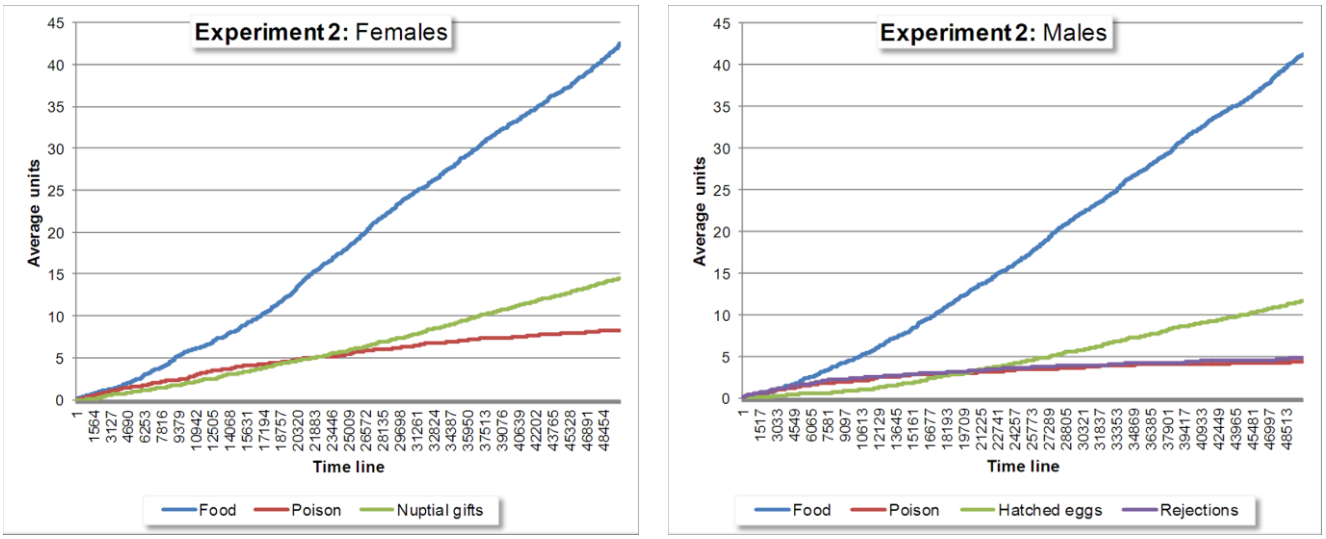


Figure 14. Experiment 2: Learning with a 1-unit-radius field of view.

males survived; and in experiment 4, 28 females and 28 males made it.

With the increment of the field-of-view radius, the policy network rapidly adjusted its weights in the second-order connections corresponding to positive-rewarding objects. However, we discovered a lack of interaction with poison in males of experiment 4 whose corresponding value network weights stayed in a range of [-0.09, 0].

Not tasting poison was beneficial for animats. Nonetheless, when poison was the first object the animat interacted with, further encounters with food sources located near poison units prevented the agent from learning how to satisfy *hunger* by feeding. We called this situation the "*first-object-tasted*" effect.

The "*first-object-tasted*" effect made creatures dependent on finding food sources that only bloomed in poison-free areas. Such effect explains the casualties registered in males of the control group in experiments 3 and 4, and explicates the lack of interaction with poison when food was the first unit that the animat touched.

Females experienced a decrement of 2.1 units of poison consumption with the extension of the field-of-view radius. Their population size benefited from their smaller outputs signals in comparison to males; females' slower response reduced the side

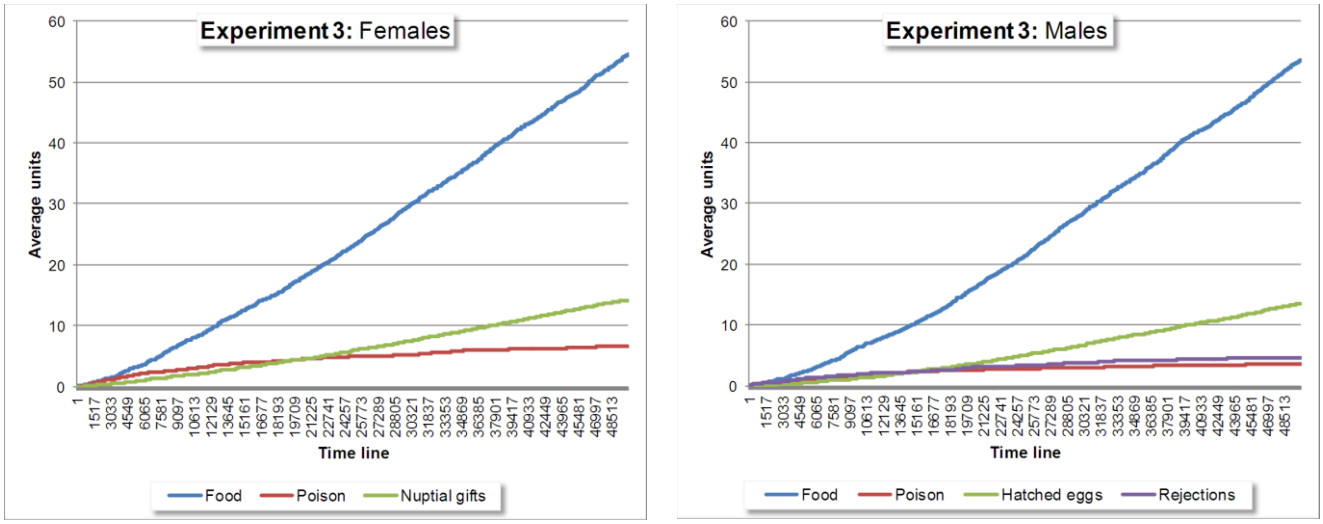


Figure 15. Experiment 3: Learning with a 1.5-unit-radius field of view.

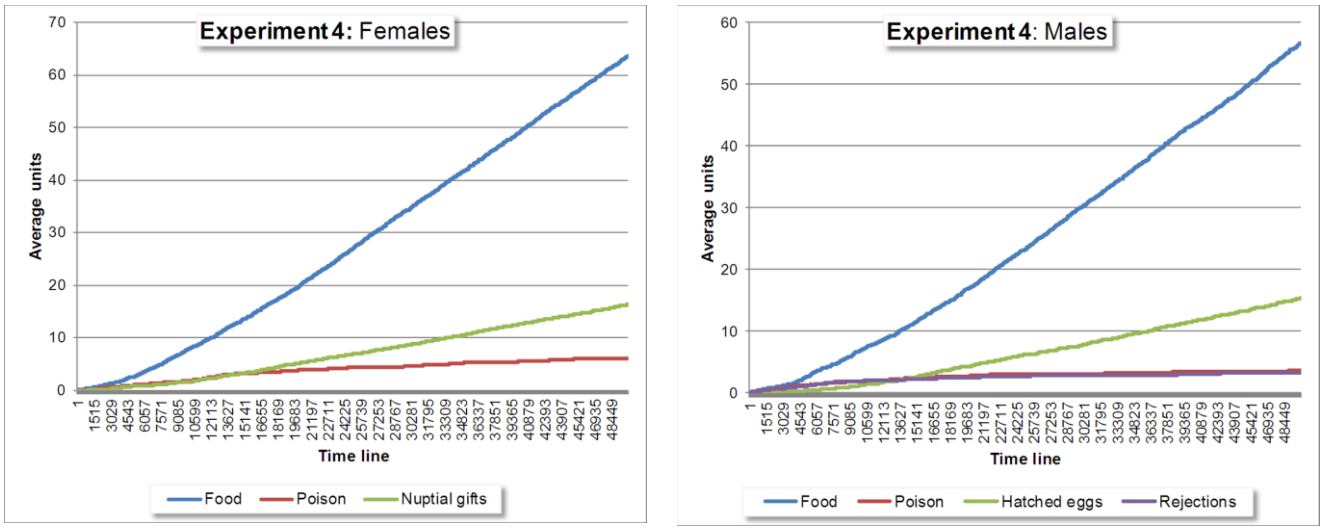


Figure 16. Experiment 4: Learning with a 2-unit-radius field of view.

effect of avoiding food sources located near poison objects.

Nuptial feeding increased 1.9 units from experiment 2 to 4, accompanying an increment of 3.6 in the amount of hatched eggs. Moreover, the number of dead eggs reduced from 3.1 to 1 in average, showing that males needed three times fewer interactions with frozen nests when they were in their brooding stage.

Females rejecting males decreased 1 unit in average from experiments 2 to 3, and from experiments 3 to 4. The "*first-object-tasted*" effect kept males from getting close to females when they did not have a nuptial gift for mating. But, in general, males preferred food sources over females when both inputs were active in their MAXSON architecture. This preference disproportion was a consequence of the faster convergence of the policy network second-order connections that related food sensors, in contrast to others external inputs'. The convergence rate imbalance increased with the field-of-view radius and caused the lack of rejections we observed in the reported results.

The mobility of females further accentuated the lower number of rejections. The decrement in the number of intercourses because of animats motion impacted the overall performance of agents in their reproduction task. Creatures' mobility kept reproduction from following the feeding tendency detected in the survival task.

We decided to take the field-of-view radius of 1.5 units for carrying on the rest of the experiments because:

a) The "*first-object-tasted*" effect did not reduce males' interaction with poison and females when having no food gift: rejections stayed around 3.5, and poison consumption was around 4 units between experiment 2 and 3.
b) It had the same benefits in the females' group survival as in the case of experiment 4: food consumption average was 54.5 units, and poison interaction was 6.5 units.

Comparing results from experiment 3 and 1, learning with the MAXSON architecture:

a) Helped 90% of initial animats survive the entire simulation by soaring food consumption and limiting interaction with poison.
b) Allowed females to receive 5 times more nuptial gifts from males.
c) Reduced rejections by half.
d) Increased the number of hatched eggs by a factor of 4.

We detected the appearance of social, animal-like behaviors after animats' policy network weights converged to 0 or 1. These behaviors started around the 20,000th time step:

a) Males pursued females when *libido* was the maximally responding filtered goal in males' policy network (figure 17a).
b) Males with higher *libido* than *hunger* lost interest in food when the attraction for a nearby female overcame the food external input signals (figure 17b).

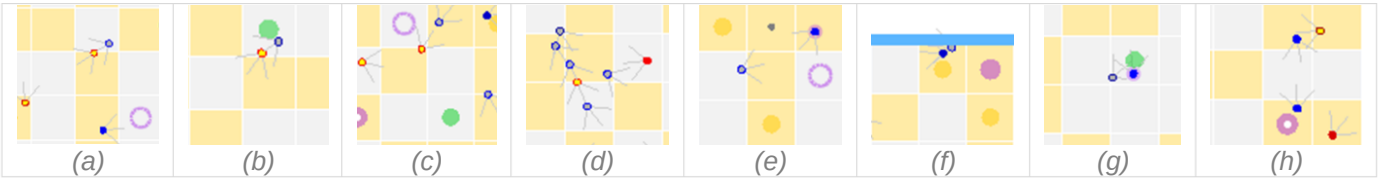


Figure 17. Behaviors that learning brought up during simulations.

c) Females that learned from nuptial-feeding followed males when no food sources were at sight (figure 17c). Females going after males immediately lost interest in their potential mate when they detected a nearby food source.
d) Males pursuing females, and females going after males, created formations we called "chains" (figure 17d). These formations lasted until the female turned back, detected her mate (both facing one another), and reproduced; or until any of the members interacted with an inert object and veered in a different direction.
e) Brooding males even turned towards poison in order to avoid frozen nests that would kill their offspring (figure 17e).
f) Animats developed avoidance forces that caused them to collide with the boundary walls (figure 17f).
g) Males carrying eggs showed opportunism when no nest was in sight and they were hungry (figure 17g).
h) Males without nuptial gifts learned to steer away from females with eggs in order to avoid rejections (figure 17h).

6.3.3. Potentiating Females' Receptivity

Females' *hunger* goal made them eat throughout the entire simulation. Females' *energy level* was mostly above 80% of their *energy capacity* when males approached with a nuptial gift. Mating females thus registered positive rewards that were smaller than 0.001, generating imperceptible changes in the value and policy second-order connection weights with respect to nuptial feeding. Consequently, females preferred food sources over males if both entities were in sight, regardless of the activation of the external *Male with Gift* input node.

We made males more valuable to females in experiment 5. Females' food sensors were disabled for 10 times longer than their usual inactivity when females turned 20,000 time steps of age. The intention of the experiment was to allow females to register larger positive external rewards than 0.001 from mating. The reinforcement increment would help females identify males with nuptial gifts as entities that satisfy *hunger*, in the same way they associate food sources with the replenishment of *energy level*.

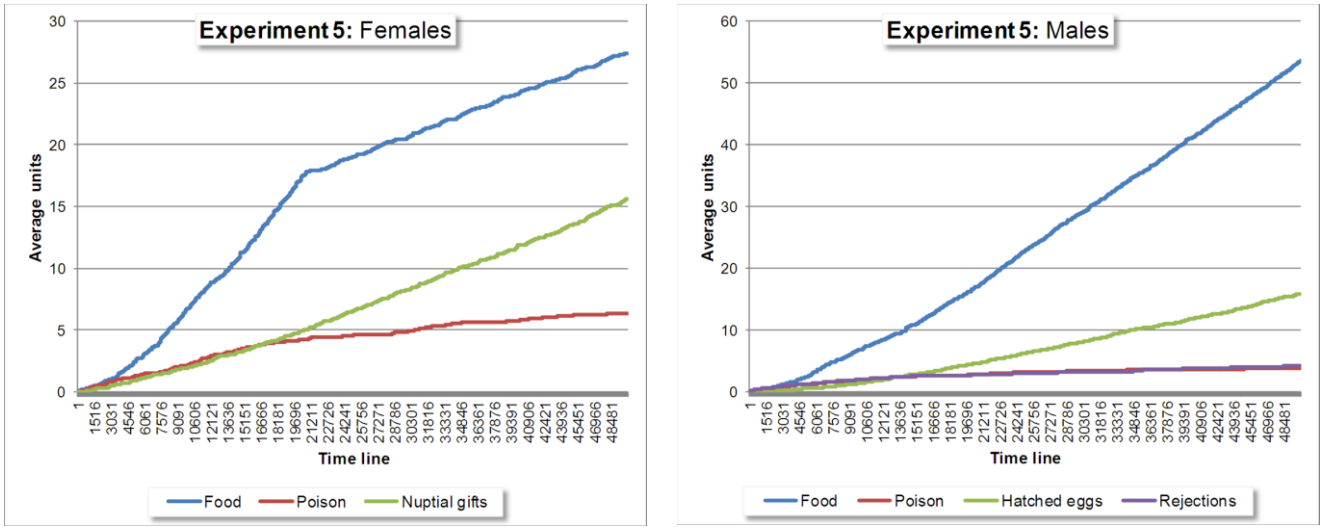


Figure 18. Experiment 5: Potentiating females' receptivity.

The environment started with 30 males and 30 females using inverted parabolic activation in their external input nodes as before.

Agents were not allowed to spawn new creatures when eggs were left in warm nests. Half of the females were born with an egg, and half of frozen and warm nests began with their respective evidence. The environment dynamically shifted the cells' temperature every 5,000 time steps to emulate heat flux and perform the evidence *cleaning process*.

Figure 18 displays the performance of the population during 50,000 time steps.

The gathered data showed a large reduction of 26.7 units in females' food consumption by the end of the simulation with respect to experiment 3. The feeding frequency started to decline when females turned 20,000 time steps of age (figure 18). However, they were still able to fulfill their survival task by feeding when their food-sensing timeout ended or by nourishing themselves with males' nuptial gifts when mating.

Disabling food sensing for a longer period did not increase nuptial feeding beyond 11% with respect to experiment 3. Nonetheless, females were more receptive toward males who tried to catch them. Such females' interest in males raised both nuptial gifts and hatched eggs by 2.1 and 2.4 units respectively.

The underlying cause for the small increment in nuptial feeding was the mobility of the living creatures. Females depended on males' nuptial gifts when they could not sense food sources as usual; however, the limited increment in nuptial gift consumption was due to the difficulty of first finding a male and later positioning in front of him, so that both would see each other and mate.

6.3.4. A More Dynamic Environment

Experiment 6 focused on the evaluation of the effects of a faster changing environment on the animats' performance in comparison with experiment 5.

As before, the ecosystem began with 30 males and 30 females that did not spawn new creatures when eggs were deposited in warm nests. Half of females were born with an egg for mating, and half of frozen and warm nests started with their respective evidence.

Again, animats used parabolic functions for activating input neurons and females disabled food sensing for 10 times longer than usual when they turned 20,000 time steps of age. Unlike experiment 5, the environment in experiment 6 shifted cells' temperature every 2,000 time steps. The purpose of the faster temperature change was to trim the exposure of evidence, so that males had less time for learning the effects of the two types of nests on their eggs.

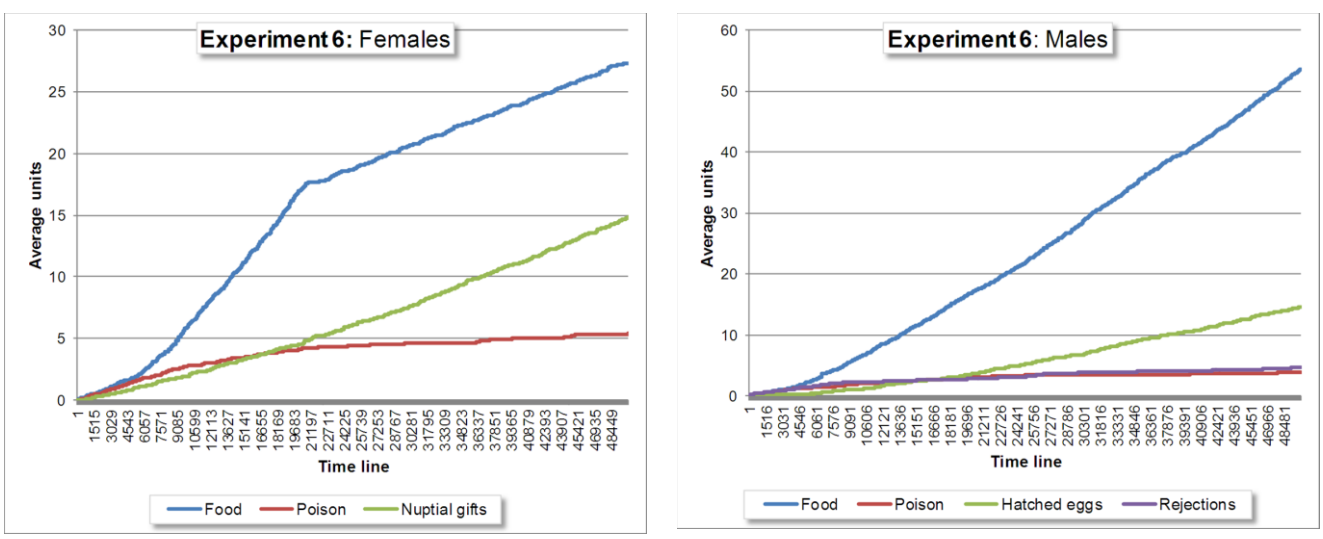


Figure 19. Experiment 6: A more dynamic environment.

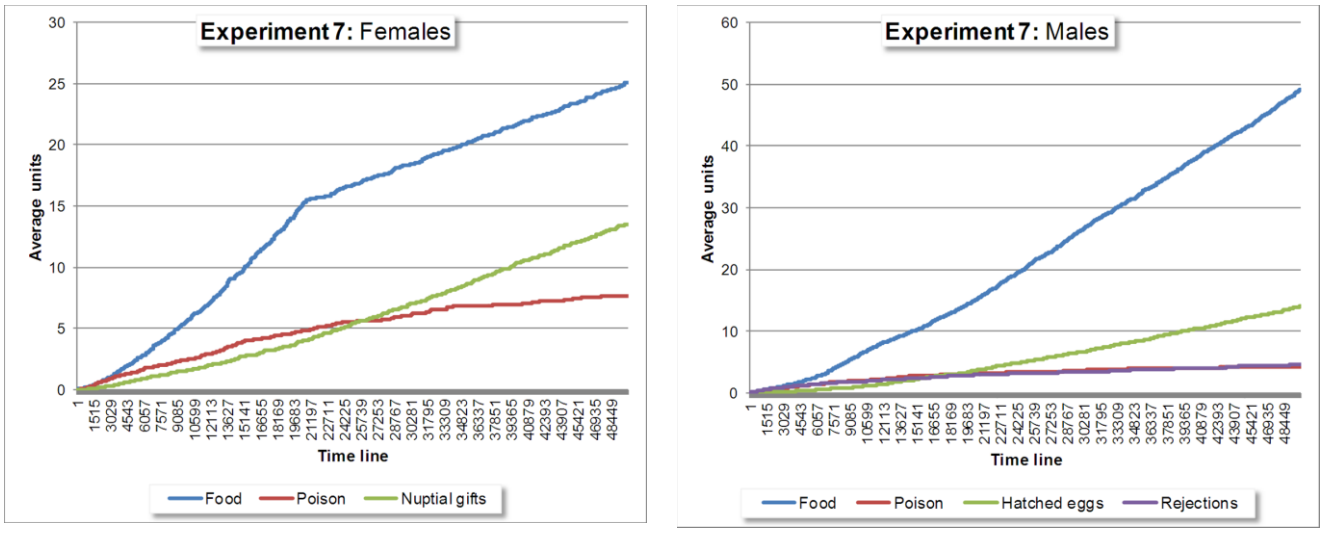


Figure 20. Experiment 7: Testing ramp functions for activating the external input neurons.

Figure 19 shows the results of the population's average performance up to the 50,000th time step.

From the data collected, we did not observe changes greater than 1.5 units in nuptial gift consumption or eggs hatched with respect to experiment 5. The results indicate that agents needed the same number of interactions with eggshells and corpses, regardless of the 60% reduction in evidence exposure.

However, male population reduced from 26 (in experiment 5) to 24 individuals by the end of the simulation. We attributed the casualties to the fast-pace swap between poison and food sources, which quickly depleted the agents' energy reservoir as a consequence of two situations:

a) It decreased the probability of interaction with food during the first 5,000 time steps. At this early stage of learning, the agent's policy second-order connections weights were still their initial values, and the animat was completely dependent on interactions that resulted from moving mostly in a straight line.
b) Animats could not avoid consuming poison because their policy weights were not yet adjusted to generate a quick response.

6.3.5. External Input Nodes Activation

In experiment 7, we show that the current inverted parabola activation function optimizes the agents' performance in comparison to the implementation of a linear function.

We set the initial configuration of the system as in experiment 5. The environment started with 30 males and 30 females, who were not allowed to spawn their offspring when the eggs were left in warm nests. Half of females' population was born with eggs, and half of nests were initialized with their respective evidence.

The environment shifted cells' temperature every 5,000 time steps for simulating the evidence *cleaning process*. Females disabled food sensing for 10 times longer than usual when they turned 20,000 time steps of age.

Each of the 60 animats had ramp functions ($f(x)=1-x$, where x is the normalized distance to the target) for activating the external input neurons of their MAXSON circuitry. The results of the experiment are displayed in figure 20.

We observed that with ramp functions, the value networks for both males and females had negative-rewarding weights that were 1.5 to 3 times larger than when we assigned them quadratic functions. These larger negative weights made evident an increment in the interaction with poison and in experiencing rejections. In contrast to experiment 5, poison consumption increased by 1.5 units in females, and males' average of poison interactions and rejections incremented 0.6 units.

There was also a reduction in the number of positive-rewarding experiences: food eaten decreased 2.3 units in females and 4.2 in males,

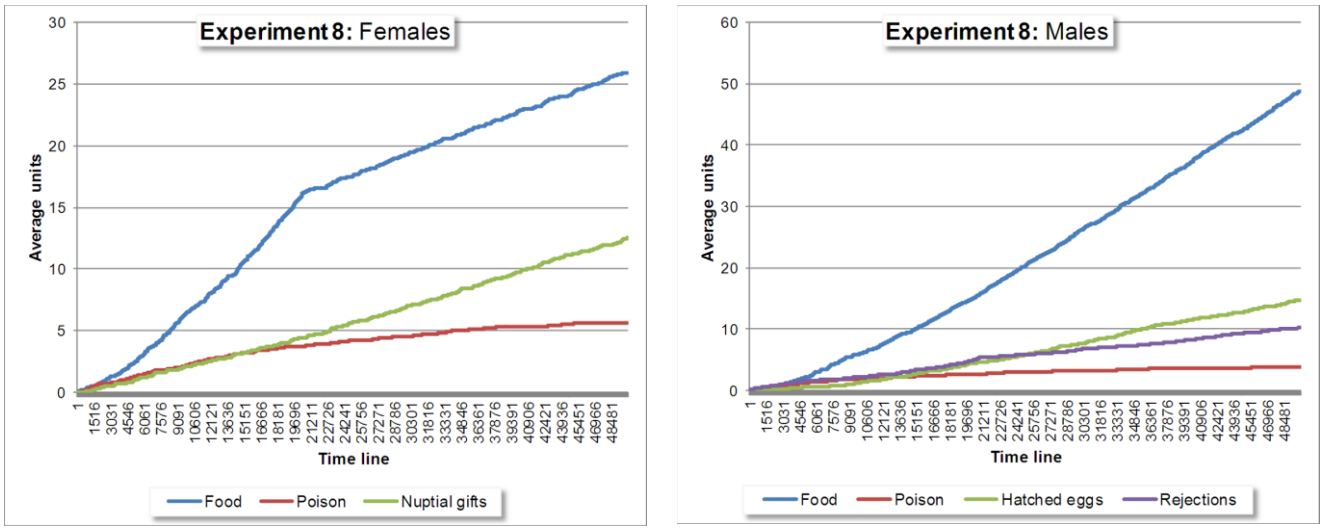


Figure 21. Experiment 8: Measuring the effects of a large population in the survival and reproduction tasks.

nuptial gifts decreased 2.1 units, and hatched eggs decreased 1.8 units.

The agents' performance worsened because the linear function yielded smaller input nodes' values than those computed with the inverted parabola of experiment 5. Consequently, the animat's policy network took longer to adjust its weights, and fed its output neurons with smaller input signals. The resulting reduced outputs' activation produced slower responses (e.g. smaller angles) that hindered poison avoidance and contributed to the increment of rejections.

6.3.6. Spawning New Creatures

Finally, we allowed agents to spawn new creatures when males left eggs in warm nests. The objective of experiment 8 consisted in evaluating the effects of a large population in the ecosystem.

The environment began with 30 males and 30 females that we tracked throughout the simulation. Half of females were born with an egg for mating, and half of frozen and warm nests started with their respective evidence. Animats used parabolic functions for activating input neurons, and females disabled food sensing for 10 times longer than usual when they turned 20,000 time steps of age. The environment shifted the cells' temperature every 5,000 time steps in order to perform the evidence *cleaning process*.

Spawning new creatures stopped at the 15,000th time step to avoid running into a slow simulation. When newborns came to life, they were fully developed and could perform all of the activities implied in the survival and reproduction tasks. Figure 21 provides graphical information on the gathered results.

Comparing experiment 5's data with the new information collected, we realized that:

a) Females reduced the average food consumption by 1.5 units, poison tasting by 0.7 units, and nuptial gifts by 3.1 units.
b) In males, the average number of food consumptions reduced by 5.2 units, while poison interactions presented no changes. The amount of hatched eggs decreased by 1.1 units, and rejections from approaching a female when having no nuptial gift multiplied 2.5 times.

The total population at the end of the simulation was 87 females and 70 males, with 2 and 8 casualties respectively. The population followed an exponential growth that was much smoother in females than in males. The final genders' population size differed because males had two goals affecting their behavior at any time: *hunger*, and one of *libido* or *brood*. We observed that *libido*, in particular, pushed males to leave areas with food sources as they were in pursuit of females for mating. Consequently, males did not stay close to food units like females, relying more on random exploration for complying with their survival task.

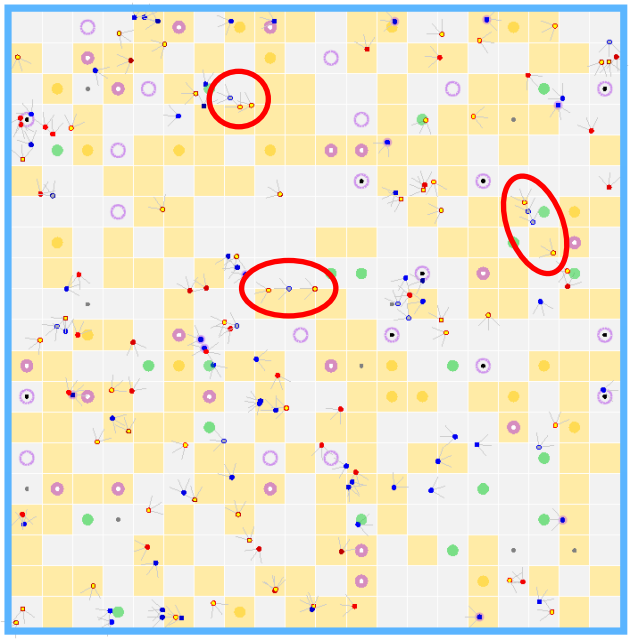


Figure 22. Chain formations.

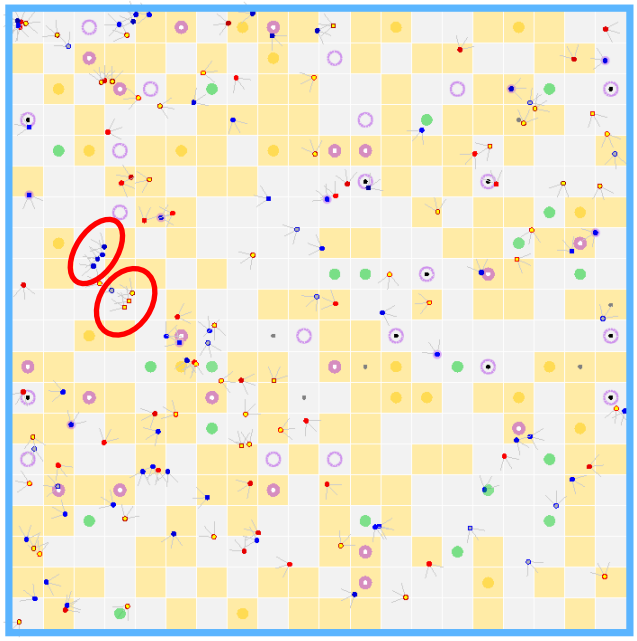


Figure 23. Shoulder-to-shoulder formations.

As the population grew, creatures became obstructions between their neighbors and the inert objects on the terrain. The lack of accessibility to inert units had a greater impact on reproduction than on survival since it caused a chain reaction that reached the social scope: males were unable to pick up nuptial gifts for mating, females rejected more males, the frequency of nuptial feeding decreased, and fewer births occurred. This effect on the reproduction task exhibited the interdependence among the elements of the environment, which was consistent to what is also observed in natural ecosystems.

Animats in the Antarctica ecosystem complied with the survival task, ensuring that their eggs hatch, despite the increment of rejections. The social interactions between agents allowed us to observe behaviors and spatial distributions that did not occur in previous experiments. The following behaviors were a consequence of having a large population of living creatures inhabiting the same space:

a) Frequent chains of males and females derived from males with *libido* as their maximally responding goal, and females pursuing males that carried nuptial gifts (figure 22).
b) Steering forces for collision avoidance created shoulder-to-shoulder formations in open areas (e.g. where there were no other inert or living entities) (figure 23).
c) Animats reproduced more frequently in the southern terrain where most of the nuptial gifts were created. Figure 24 shows females with eggs exploring the northern environment, while most of the females to the south do not have an egg because they have just copulated with a male.

The attached appendix contains the configuration of a male and female brain at the end of experiment

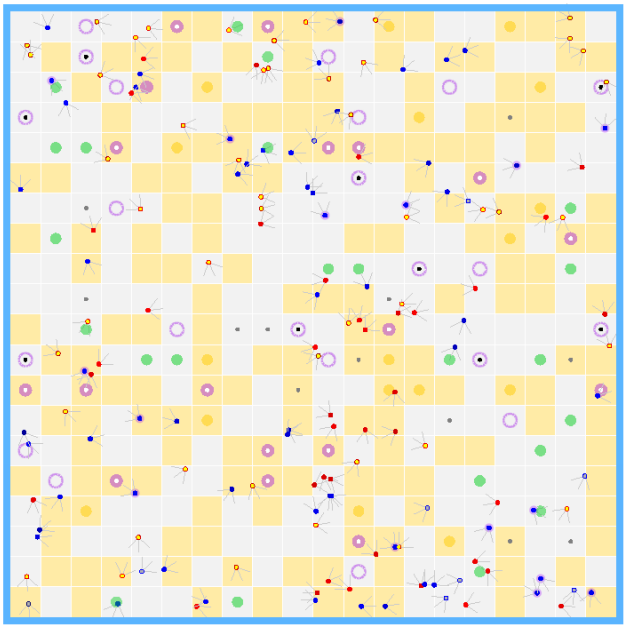


Figure 24. Spatial distribution of the nuptial gifts determined areas where reproduction was more likely to happen.

8's simulation. It also includes C++ code that corresponds to the MAXSON circuitry, the agent class, and the "simulate" loop function of the Antarctica ecosystem.

7. Conclusion

We have demonstrated that the MAXSON architecture is a flexible learning framework for modeling animal behavior. In particular, the creatures in the Antarctica ecosystem were able to:

- Survive by eating food and avoiding poison.
- Reproduce by nuptial-feeding during mating, and by carrying their offspring to warm nests for hatching.

Simulations, like the one presented, can be further improved in their level of accuracy with biology-inspired features. Enhanced artificial ecosystems thus can become potential tools, which in the spirit of ecology, should be used for studying and understanding animal behavior and the bonds within the environmental components [Dorin 2008].

8. Previous Work

Crabbe and Dyer [Crabbe 2001] introduced the MAXSON architecture. Their neural architecture was tailored specifically for animats, which could not make use of traditional reinforcement learning methods because of their restrictively large number of iterations required for learning. Their seminal work consisted of simulating agents that learned to survive by eating food, drinking water, and avoiding poison. In later developments, they showed the integration of reactive behavior and vicarious learning through an extension of the original model that they called VI-MAXSON [Crabbe 2000a]. Their new structure aimed to favor inexperienced animats through the perception of other agents' state without the need for direct interaction with the objects.

Extending the scope of the MAXSON architecture, [Crabbe 2000b] delved into goal-sequence learning, under the VI-MAXSON schema, for solving a wall-building problem. The development in [Crabbe 2000b] was an enhancement to the second-order networks explicitly designed for goal-sequencing in [Crabbe 1999]. VI-MAXSON allowed a student animat to repeat a succession of goals by observing a teacher performing the-wall building task.

The Antarctica ecosystem implements the original MAXSON architecture with an additional filtering network. Our extended neural model allows managing multiple goals differently, according to animat's sex or stage of life. Our MAXSON approach is inspired by recent experimentation on laboratory mice that gives evidence to the existence of both male and female neural circuitry in females. Male behavior triggers in response to the lack of olfactory readings after suppressing the channels that feed such structure in the female's brain [Kimchi 2007].

To our knowledge, the Antarctica ecosystem is the first application that models *nuptial feeding* [Elliot 1998] and *male brooding* [Williams 1995]. Our simulation mimics a reproduction task upon which animats ensure the survival of their species, without need for an explicit evolutionary system.

9. Acknowledgement

The development of this project was made possible by the financial support of the *Consejo Nacional de Ciencia y Tecnología* (CONACYT) [Mexican National Council for Science and Technology] through the Graduate Fellowship to Study Abroad program.

Special thanks to Prof. Michael G. Dyer for his advice, and to Joyce Kuo for her help in organizing and polishing the content of the present report.

10. References

[Beer 1990] Beer, R. D. (1990). *Intelligence as Adaptive Behavior: An Experiment in Computational Neuroethology*. Academic Press.

[Buckland 2005] Buckland, Mat. (2005). *Programming Game AI by Example*. Wordware Publishing, Inc.

[Crabbe 1999] Crabbe, Frederick L.; and Dyer, Michael G. (1999). *Second-Order Networks for Wall-Building Agents*. Proceedings of the International Joint Conference on Neural Networks (IJCNN-99).

[Crabbe 2000a] Crabbe, Frederick L.; and Dyer, Michael G. (2000). *Goal Directed Adaptive Behavior in Second-Order Neural Networks: The MAXSON Family of Architectures*. Adaptive Behavior 8(2).

[Crabbe 2000b] Crabbe, Frederick L.; and Dyer, Michael G. (2000). *Observation and Imitation: Goal Sequence Learning in Neurally Controlled Construction Animats: VI-MAXSON*. In: J.-A. Meyer, A. Berthoz, D. Floreano, H. Roitblat and S. W. Wilson (eds.) From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior. Bradford Book/MIT Press.

[Crabbe 2001] Crabbe, Frederick L.; and Dyer, Michael G. (2001). *Goal Directed Adaptive Behavior in Second-Order Neural Networks: Learning and Evolving in the MAXSON Architecture*. In: M. J. Patel, V. Honavar, and K. Balakrishnan (eds.) Advances in the Evolutionary Synthesis of Intelligent Agents. Bradford Books/MIT Press.

[Dorin 2008] Dorin, Alan; Korb, Kevin B.; and Grimm, Volker. (2008). *Artificial-Life Ecosystems: What are they and What Could They Become?* Artificial Life XI, Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems. MIT Press.

[Elliot 1998] Elliott, Lang; and Read Marie. (1998). *Common Birds and Their Songs*. Houghton Mifflin Harcourt.

[Hill 2001] Hill Jr., Francis S. (2001). *Computer Graphics Using OpenGL*. Second Edition. Prentice Hall.

[Kimchi 2007] Kimchi, Tali; Xu, Jennings; and Dulac, Catherine. (2007). *A functional circuit underlying male sexual behaviour in the female mouse brain*. Nature, Vol. 448(30).

[Mrosovsky 1980] Mrosovsky, N; and Sherry, D. F. (1980). *Animal Anorexias*. Science, Vol. 207(4433).

[Sutton 1991] Sutton, Richard S. (1991). *Reinforcement Learning Architectures for Animats*. Proceedings of the International Workshop on the Simulation of Adaptive Behavior: From Animals to Animats. MIT Press.

[Sutton 1998] Sutton, Richard S.; and Barto, Andrew G. (1998). *Reinforcement Learning*. MIT Press.

[Svensson 1987] Svensson, Bo G.; and Petersson Erik. (1987) *Sex-Role Reversed Courtship Behaviour, Sexual Dimorphism and Nuptial Gifts in the Dance Fly, Empis Borealis*. Ann. Zool. Fennici 24.

[Werner 1994] Werner, Gregory M. (1994). *Using Second Order Neural Connections for Motivation of Behavioral Choices*. From Animals to Animats 3: Proceedings of the Third International Conference of Simulation of Adaptive Behavior.

[Williams 1995] Williams, Tony D. (1995). *The Penguins*. Oxford University Press.

Antarctica

Exploring the MAXSON Architecture

Appendix

A.1. Male and Female Brains

Tables 1 and 2 show the weights in the connections of the value and policy networks, which defined the configuration of the brain of a surviving male from experiment 8.

Input	Goal	Weight
Food Left	Hunger	1
	Libido	0
	Brood	0
Food Right	Hunger	1
	Libido	0
	Brood	0
Poison Left	Hunger	-0.09737
	Libido	-0.09735
	Brood	-0.09725
Poison Right	Hunger	-0.29208
	Libido	-0.29201
	Brood	-0.29173
Warm Nest Left	Hunger	0
	Libido	0
	Brood	0.96736
Warm Nest Right	Hunger	0
	Libido	0
	Brood	1
Frozen Nest Left	Hunger	0
	Libido	0
	Brood	-0.58019
Frozen Nest Right	Hunger	0
	Libido	0
	Brood	-0.29004
Female & Gift Left	Hunger	0
	Libido	1
	Brood	0
Female & Gift Right	Hunger	0
	Libido	1
	Brood	0
Feale & No Gift Left	Hunger	0
	Libido	-0.66729
	Brood	0
Female & No Gift Right	Hunger	0
	Libido	-0.84939
	Brood	0
Food Gift Left	Hunger	0
	Libido	1
	Brood	0
Food Gift Right	Hunger	0
	Libido	0.98732
	Brood	0

Male with Gift Left	Hunger	0
	Libido	0
	Brood	0
Male with Gift Right	Hunger	0
	Libido	0
	Brood	0

Table 1. First-order value network weights in a male's brain.

Input	Output	Hunger's Weights	Libido's Weights	Brood's Weight
Food Left	Turn Left	1	0.50553	0.50450
	Turn Right	0.28480	0.50239	0.49511
Food Right	Turn Left	0	0.49759	0.50075
	Turn Right	0.94582	0.50123	0.50508
Poison Left	Turn Left	0.45506	0.13372	0.21222
	Turn Right	0.79845	1	0.71231
Poison Right	Turn Left	0.50316	1	1
	Turn Right	0.11852	0.19784	0.15389
Warm Nest Left	Turn Left	0.50518	0.49714	1
	Turn Right	0.49696	0.49551	0.28879
Warm Nest Right	Turn Left	0.49645	0.49826	0.10811
	Turn Right	0.49611	0.49480	0.99702
Frozen Nest Left	Turn Left	0.50279	0.49505	0
	Turn Right	0.49504	0.49546	1
Frozen Nest Right	Turn Left	0.50137	0.50466	1
	Turn Right	0.49809	0.50125	0.35737
Female & Gift Left	Turn Left	0.49689	1	0.50395
	Turn Right	0.50133	0	0.49847
Female & Gift Right	Turn Left	0.49618	0.03373	0.49881
	Turn Right	0.49875	1	0.50434
Feale & No Gift Left	Turn Left	0.50286	0	0.49522
	Turn Right	0.49512	0.98199	0.50577
Female & No Gift Right	Turn Left	0.50221	1	0.49657
	Turn Right	0.50082	0	0.50584
Food Gift Left	Turn Left	0.50297	0.87985	0.50484
	Turn Right	0.50202	0	0.50029
Food Gift Right	Turn Left	0.50128	0.50320	0.49539
	Turn Right	0.49830	1	0.50171
Male with Gift Left	Turn Left	0.50066	0.49781	0.50276
	Turn Right	0.50371	0.49578	0.49465
Male with Gift Right	Turn Left	0.50487	0.49531	0.49454
	Turn Right	0.50485	0.49735	0.50185

Table 2. Second-order policy network weights in a male's brain.

Tables 3 and 4 show the weights in the connections of the value and policy networks, which defined the configuration of the brain of a surviving female from experiment 8.

Input	Goal	Weight
Food Left	Hunger	1
	Libido	0
	Brood	0
Food Right	Hunger	1
	Libido	0
	Brood	0
Poison Left	Hunger	-0.39032

	Libido	0
	Brood	0
Poison Right	Hunger	-0.29292
	Libido	0
	Brood	0
Warm Nest Left	Hunger	0
	Libido	0
	Brood	0
Warm Nest Right	Hunger	0
	Libido	0
	Brood	0
Frozen Nest Left	Hunger	0
	Libido	0
	Brood	0
Frozen Nest Right	Hunger	0
	Libido	0
	Brood	0
Female & Gift Left	Hunger	0
	Libido	0
	Brood	0
Female & Gift Right	Hunger	0
	Libido	0
	Brood	0
Feale & No Gift Left	Hunger	0
	Libido	0
	Brood	0
Female & No Gift Right	Hunger	0
	Libido	0
	Brood	0
Food Gift Left	Hunger	0
	Libido	0
	Brood	0
Food Gift Right	Hunger	0
	Libido	0
	Brood	0
Male with Gift Left	Hunger	0.68502
	Libido	0
	Brood	0
Male with Gift Right	Hunger	0.61567
	Libido	0
	Brood	0

Table 3. First-order value network weights in a female's brain.

Input	Output	Hunger's Weights	Libido's Weights	Brood's Weight
Food Left	Turn Left	1	0.49635	0.49474
	Turn Right	0.28601	0.49639	0.50119
Food Right	Turn Left	0.45612	0.49822	0.50352
	Turn Right	0.97074	0.50545	0.49720
Poison Left	Turn Left	0.05520	0.50578	0.49843
	Turn Right	0.90091	0.49595	0.49855
Poison Right	Turn Left	0.95159	0.49753	0.50352
	Turn Right	0.27824	0.49977	0.49462
Warm Nest Left	Turn Left	0.49778	0.49492	0.49593
	Turn Right	0.49933	0.50149	0.50530
Warm Nest Right	Turn Left	0.50016	0.49865	0.50297

	Turn Right	0.49911	0.50459	0.49452
Frozen Nest Left	Turn Left	0.49715	0.50352	0.50078
	Turn Right	0.49802	0.49802	0.50409
Frozen Nest Right	Turn Left	0.50052	0.49453	0.50383
	Turn Right	0.49809	0.50162	0.49438
Female & Gift Left	Turn Left	0.50596	0.49643	0.49971
	Turn Right	0.49948	0.49410	0.49858
Female & Gift Right	Turn Left	0.50260	0.49446	0.49839
	Turn Right	0.49502	0.49704	0.49478
Feale & No Gift Left	Turn Left	0.50260	0.50066	0.49873
	Turn Right	0.49729	0.50593	0.50342
Female & No Gift Right	Turn Left	0.49547	0.50033	0.50245
	Turn Right	0.50173	0.49747	0.49849
Food Gift Left	Turn Left	0.49738	0.49939	0.49533
	Turn Right	0.50211	0.50434	0.50211
Food Gift Right	Turn Left	0.50177	0.50204	0.50214
	Turn Right	0.49717	0.49877	0.50269
Male with Gift Left	Turn Left	1	0.49917	0.49955
	Turn Right	0.50135	0.49855	0.50433
Male with Gift Right	Turn Left	0.50461	0.49941	0.50530
	Turn Right	0.83805	0.49708	0.49908

Table 4. Second-order policy network weights in a female's brain.

A.2. Animat Class

The following C++ code defines the Animat class most relevant functions. The actual implementation includes calls to OpenGL and Eigen Libraries for drawing and linear algebra computations respectively.

```
/*********************** Constructor **************************/
Animat::Animat(int i, const Vector3d& pos, int sx, const Genotype& g)
{
	id = i;
	//Id for this animat.
	setPosition(pos);		//Set initial position.
	genotype = g;			//Copy the genetic material into its own.
	maxSpeed = genotype.getMaxSpeed(); //Max speed in world units per simulated second.
	facing = Vector3d::Random();
	facing[2] = 0;
	facing.normalize();		//Generate a random facing vector.
	velocity = facing * maxSpeed;	//Generate new velocity.
	personalRadius = 1.0/Calculus::getMainScale();		//Personal radius.
	temperatureRadius = 3.0/Calculus::getMainScale();	//Temperature radius.
	viewRadius = 1.5;		//View radius (in world units).
	sex = sx;			//Sex.
	foodCapacity = 200.0/maxSpeed;	//Food capacity.
	temperature = 1.0;		//Animat's body maximum temperature.
	speed = maxSpeed;		//Speed starts with the max speed.
	age = 0.0;			//Start age.
	maxAgeForLearning = 600.0;	//Maximum age for applying learning.
	obstacles.clear();		//Empty obstacles vector.
	hungerStep = 0.01;		//Amount to increase every time step to hunger.
	libidoStep = 0.008;		//Will be changing sign to reflect direction.
	carryingEgg = 0;		//Initially no animat is carrying an egg.
	carryingEgg_1 = 0;
	carryingGift = 0;		//Initially no animat is carrying a gift.
	carryingGift_1 = 0;
	partnerGenotype = NULL;		//No partner is associated at birth.
	growingEggTime = 0;		//Set a timer for a female to have an egg.
```

```
	rewardGift = false;
	receivedGift = false;		//For females receiving a gift from a male.
	brain.setSex(sex);		//Pass information to brain for decision-making.

	if(sex == 1 && Calculus::randomGenerator(0.0, 1.0) > 0.5)
		growingEggTime = floor(foodCapacity * 20);//Initialize some of the females.

	for(int I=0; I<Brain::totalGoals; I++)		//Set current state to 0.
		goals[I] = 0.0;	//This way it is synchronized with brain goals array too.

	for(int I=0; I<Brain::totalObjects; I++)	//Set waiting time for objects.
		senseAgain[I] = 0;

	for(int I=0; I<Brain::totalInputs; I++)		//Initialize the gift inputs.
		igift[I] = igift_angle[I] = igift_closeness[I] = 0.0;

	foodEaten = poisonEaten = laidEggs = birthsObserved = deadsObserved = gifts =
		rejections = 0;				//Bookkeeping variables.
	track = false;					//Do not track any animat, initially.

}

//////////////////// Functions for reward and punishment implementation ////////////////////

/*** Special function to perform hurting (like from eating poison) ***/
void Animat::hurt(int by)
{
	//Define what caused the pain.
	switch(by)
	{
		case Brain::Poison:
			{
				for(int I=0; I<Brain::totalGoals; I++)	//Animat can be hurt by
				{	//poison (at any state) and affect all of the goals.
					//Everything is based on foodCapacity.
					goals[I] += 5.0*(Brain::theta2 + hungerStep/foodCapacity);
					//Truncate values (Except for Brood goal).
					if(goals[I] > 1.0 && I != Brain::Brood) goals[I] = 1.0;
					if(sex == 1 && (I==Brain::Libido || I==Brain::Brood))
						goals[I] = 0.0;//Females have no libido or brood.
				}

				senseAgain[Brain::Poison] = -1;
				poisonEaten++;
				displayVariables();
				break;
			}
		case Brain::Female:
			{
				goals[Brain::Libido] += 5.0*(Brain::theta2 +
					hungerStep/foodCapacity);
				//Truncate to 1.
				if(goals[Brain::Libido] > 1.0) goals[Brain::Libido] = 1.0;
				senseAgain[Brain::Female] = -1;
				rejections++;
				displayVariables();
				break;
			}
		case Brain::FrozenNest:		//Animat will be affected by frozen nest when it
						//observes a dead baby animat in it.
			{
```

```
                        goals[Brain::Brood] += 5.0*(Brain::theta2 +
                               hungerStep/foodCapacity);
                        //Not necessary to truncate to one because it has its own flag
                        //(carryingEgg) that prevents using values
                        //out of boundary in the setGoals procedure in brain.
                        senseAgain[Brain::FrozenNest] = -1;
                        deadsObserved++;
                        displayVariables();
                        break;
                }
        }
}

/**** Function to simulate the act of replenishing food capacity ****/
void Animat::feed()
{
        goals[Brain::Hunger] = 0.0;        //Reduce hunger.
        senseAgain[Brain::Food] = -1;      //Mark it for avoiding sensing food in the next
                                           //time steps.
        foodEaten++;
        displayVariables();
}

/***** Function to simulate when an agent has approached a nest *****/
bool Animat::bumpNest(Nest *nest)
{
        if(sex == 0)        //Only males can lay eggs received from females.
        {
                //First, check what is in the nest.
                if(nest->isThereABlackEgg())      //Found a dead baby?
                {
                        hurt(Brain::FrozenNest);    //The state of the nest indicates
                                                    //that it is frozen to cause a baby's death.
                        return true;                //Done this time step.
                }                                   //Even if it was carrying an egg, it
                                                    //realizes it should not drop it in here.
                else
                {
                        bool valueToReturn = false;

                        if(nest->isThereAWhiteEgg())      //Found a hatched egg?
                        {
                                goals[Brain::Brood] -= 5.0*(Brain::theta2 +
                                       hungerStep/foodCapacity);   //Reduces goal to create a
                                                                   //positive reward.
                                senseAgain[Brain::WarmNest] = -1;  //It is already assumed that
                                                                   //only warm nests can have
                                                                   //positive reward.
                                birthsObserved++;
                                displayVariables();
                                valueToReturn = true;       //Completed a task this time step.
                        }

                        if(carryingEgg == 1)        //Lay an egg only if it was in a
                                                    //parenting state, either in a frozen or
                                                    //warm nest.
                        {
                                if(partnerGenotype == NULL) //Checkpoint that partnerGenotype is
                                                            //not NULL.
                                {
                                        assert(false);      //End program.
```

```
                        }

                        updateCarryingEgg(0);       //Disable parenting state.
                        if(!nest->isFrozen())
                        {
                               laidEggs++;
                               displayVariables();
                        }

                        //Since it left an egg, disable its ability to sense nests for a
                        //while.
                        senseAgain[Brain::WarmNest] = -1;
                        senseAgain[Brain::FrozenNest] = -1;       //Both, to allow
                                                                  //sufficient time for
                                                                  //simulating his coming
                                                                  //back.
                        senseAgain[Brain::Female] = -1;           //Avoid sensing other
                                                                  //females some time
                                                                  //steps to prevent
                                                                  //reproduction with
                                                                  //its own child.

                        return true;                              //Done this time step.
                    }

                    return valueToReturn;       //This is because it might get reward and
                                                //leave an egg at the same time step.
                }
        }       //Nothing is learned from approaching a nest when there is no need for it.
        return false; //This indicates that the agent can do something else this time step.
}

/********* Function to simulate when the animat bumps a gift *********/
bool Animat::bumpGift()
{
        if(sex == 0 && carryingEgg == 0 && carryingGift == 0)
        {
                updateCarryingGift(1);              //Now it is carrying a gift.
                senseAgain[Brain::Gift] = -1;
                return true;                        //Did something with gift.
        }
        else
                return false;               //May use time step for doing something else.
}

/*** Function to remember information about how the animat bumped a gift ***/
void Animat::rememberGift(double closenessLeft, double activationLeft, double
closenessRight, double activationRight)
{
        igift[Brain::GdL] = activationLeft;       //Set as active only these inputs and left
                                                  //the rest as zeros.
        igift_closeness[Brain::GdL] = closenessLeft;
        igift[Brain::GdR] = activationRight;
        igift_closeness[Brain::GdR] = closenessRight;
}

/******** Function to simulate reproduction / receiving an egg *******/
bool Animat::reproduce(Animat* female)
{
        if(sex == 0 && carryingEgg == 0)   //Only males can take eggs when they are not
                                           //brooding.
```

```
	{
		if(isCarryingGift())		//Does male have a gift to offer to female?
		{
			if(partnerGenotype != NULL) //Check point that male is not receiving an
								//egg when it had one previously.
			{
				assert(false);
			}

			partnerGenotype = female->deliverEggToMale(id);  //Link partnerGenotype
								//to the dynamic genotype created by female.
			goals[Brain::Libido] = goals[Brain::Libido] / 2.0; //Reduce the libido
								//urgency by half to leave credit to gift
								//rewarding.
			senseAgain[Brain::Female] = -1;	//Avoid sensing other females some
								//time steps.
			rewardGift = true;	//Activate it here and disable it when it gives the
						//food-gift.

			female->receiveGiftFromMale();	//Female is marked to be rewarded
								//when it is her turn in the list.
		}
		else
			hurt(Brain::Female);		//Hurt by proposing with no gift to offer.

		return true;			//Done this time step.
	}
	else
		return false;			//Did nothing.
}

/* Function to complete the reproduction process with a female */
void Animat::endReproduction()
{
	assert(rewardGift);		//Check that indeed, it has reproduced before.
					//This is the second phase of reproduction.
	rewardGift = false;		//Disable flag.
	updateCarryingEgg(1);		//Postpone Brood goal from going to above current max goal
					//when it implements learning.
	goals[Brain::Libido] = 0.0; //Give credit to gift.
	updateCarryingGift(0);		//No more carrying a gift, now it is an egg.
}

/*** Function to know if animat laid an egg when it bumped a nest ***/
bool Animat::droppedEgg()
{
	return (carryingEgg_1 > carryingEgg);	//Went from carrying to not carrying?
}

////////////////////////////////////////////////////////////////////////////////////////////

////////// Functions for updating the life state of the animat as time progresses //////////

/******** Function to shift the current value of carrying egg ********/
/* If during current time step the animat bumps a nest, carrying-egg */
/* value might change, e.g. function allows to detect any change in  */
/* parenting status, which is updated in the brain at the end of time*/
/* step. Must be called before doing any action to ensure propagation*/
void Animat::updateParenting()
{
	updateCarryingEgg(carryingEgg);
```

```
}

/***** Function to discount one tick to the poison waiting time ******/
void Animat::updatePoison()
{
	if(senseAgain[Brain::Poison] > 0)	//Is there a timeout for poison sensing?
	{
		senseAgain[Brain::Poison]--;
	}
}

/******* Function to update libido for males every time step *********/
void Animat::updateLibido()
{
	if(sex == 0)					//Is it a male?
	{
		if(goals[Brain::Libido] >= 0.6)		//It reached 0.5 without reproducing.
		{
			libidoStep = -foodCapacity * Brain::theta2 * 0.9;	//Now it goes
								//downward (faster).
		}
		else
		{
			if(goals[Brain::Libido] == 0.0)
				libidoStep = 0.008;		//Now it goes upward.
		}

		goals[Brain::Libido] += libidoStep / foodCapacity;
		if(goals[Brain::Libido] < 0.0)
			goals[Brain::Libido] = 0.0;		//Avoid negative numbers.
		else
		{
			if(goals[Brain::Libido] > 1.0)
				goals[Brain::Libido] = 1.0;	//Cap to upper bound.
		}
	}
}

/********** Function to set the new value of carrying egg ***********/
void Animat::updateCarryingEgg(unsigned int val)
{
	carryingEgg_1 = carryingEgg;
	carryingEgg = val;	//Keeps memory necessary for knowing when Brood goal has to get 1
				//in one time step.
}

/*********** Function to set a new value of carrying gift **********/
void Animat::updateCarryingGift(int val)
{
	carryingGift_1 = carryingGift;
	carryingGift = val;	//Keeps memory of previous carrying gift value.
}

/**** Function to shift the carrying-gift value every time step ****/
void Animat::shiftCarryingGift()
{
	updateCarryingGift(carryingGift);
}

//////////////////////////////////////////////////////////////////////////////////////////////
```

```
//////////////////////////////////// Functions for females only ////////////////////////////////

/******* Function to know when the female is ready to reproduce ******/
bool Animat::isFemaleReady()
{
	return (growingEggTime == floor(foodCapacity * 20));
}

/****** Function to update the growing egg time in the female *******/
void Animat::updateGrowingEggTime()
{
	if(sex == 1 && growingEggTime < floor(foodCapacity * 20))	//Check boundary.
	{
		growingEggTime++;
	}
}

/****** Function to set a female growing egg time when it mates ******/
Genotype* Animat::deliverEggToMale(int i)
{
	if(sex == 1)
	{
		growingEggTime = 0;  //Reset the time, so that female becomes unavailable.

		// Create a copy of her own genotype, which will be contained in the egg.
		Genotype* herCopy = new Genotype(genotype);

		return herCopy;
	}
	else
		return NULL;
}

/****** Function to know if female has received a gift from a male *******/
bool Animat::didSheReceiveGift()
{
	return receivedGift;
}

/**** Function to simulate the act of receiving the gift from a male ****/
void Animat::receiveGiftFromMale()
{
	receivedGift = true;		//Mark female, so that this/next time step she will be
					//positive-reinforced.
}

/** Function to reward female by reducing her hunger from male's gift **/
void Animat::rewardByGiftFromMale()
{
	goals[Brain::Hunger] = 0.0;		//Reduce hunger.
	gifts++;
	receivedGift = false;			//Female completed her part in mating process.
	displayVariables();
}

//////////////////////////////////////////////////////////////////////////////////////////////

//////////////////// Functions to implement learning and decision making ////////////////////

/********** Function to implement learning in this animat ************/
void Animat::implementLearning()
```

```
{
        if(carryingGift < carryingGift_1)        //Just gave its gift to female? Send memory
                                                 //from experience with gift.
                brain.setInput(igift, igift_angle, igift_closeness);//This is to assign a
                                                 //delayed reward in the value network.
        else
                brain.setInput(input, input_angle, input_closeness);//At this point, all
                                                 //inputs are stored with correct activation
                                                 //values in [0,1].

        if(age < maxAgeForLearning && brain.getInputSum() > 0.0)      //Learn only if it is
                                                                      //allowed.
        {
                // At this point, all inputs must be stored in the brain.
                brain.computeReinforcementSignals();     //Generate reinforcement signals
                                                         //based on maximal goal that brought
                                                         //the animat up to here.
                brain.adjustPolicyNetwork();             //Update weights in policy network
                                                         //based on maximal goal at previous
                                                         //time step.
                brain.setGoals(goals);                   //Copy goal internal sensor readings
                                                         //into the brain, who remembers goals
                                                         //in previous time steps.
                brain.adjustValueNetwork();              //Update weights in value network to
                                                         //detect which goal was satisfied.
        }
        else
                brain.setGoals(goals);                   //Or just let the brain know of the
                                                         //new internal goals state.

        brain.setParenting(carryingEgg);                 //Tells the brain the parenting state
                                                         //for next time step.
        if(carryingEgg > carryingEgg_1)                  //Went from 0 to 1?
        {
                goals[Brain::Brood] = goals[Brain::Hunger] + 5.0 * Brain::theta2;    //Modify
                                                         //directly the goal.
        }       //Brain must know next time step that this change occurred.

        //Disable sensing if animat was marked.
        if(senseAgain[Brain::Food] == -1)
        {
                senseAgain[Brain::Food] = foodCapacity * 3.0;
                if(sex == 1 && age > 200.0)
                        senseAgain[Brain::Food] = foodCapacity * 30.0;
        }
        if(senseAgain[Brain::Poison] == -1)
        {
                senseAgain[Brain::Poison] = foodCapacity * 3.0;
        }
        if(senseAgain[Brain::WarmNest] == -1)
        {
                senseAgain[Brain::WarmNest] = foodCapacity * 3.0;
        }
        if(senseAgain[Brain::FrozenNest] == -1)
        {
                senseAgain[Brain::FrozenNest] = foodCapacity * 3.0;
        }
        if(senseAgain[Brain::Female] == -1)
        {
                senseAgain[Brain::Female] = foodCapacity * 3.0;
        }
```

```
	if(senseAgain[Brain::Gift] == -1)
	{
		senseAgain[Brain::Gift] = foodCapacity * 3.0;
	}
}

/****** Function that determines the next motion for the animat ******/
void Animat::makeDecision()
{
	if(brain.getInputSum() > 0.0 && canDoSomethingNextTimeStep()) //Is there a
								//significant activation?
	{
		brain.makeDecision(output);	//Load new output.
		neuralSteeringForce = Calculus::rotateVector(output[Brain::TurnLeft]
			- output[Brain::TurnRight], facing);
		neuralSteeringForce *= maxSpeed;
	}
	else
		neuralSteeringForce = Vector3d::Zero();		//A neutral force.

}

////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////// Functions for steering and motion ////////////////////////////

/********** Function to simulate a random walk and return ************/
/*********************** a steering force ***************************/
Vector3d Animat::randomWalk()
{
	Vector3d steeringForce = facing.normalized();	//Initialize an homogeneous vector
	                                                //with previous velocity/facing.
	double n1 = Calculus::randomGenerator(-1.0, 1.0);
	double n3 = Calculus::randomGenerator(0.0, 1.0);

	if(n3 > 0.9)                      //Change direction certain percentage of the time.
	{
		steeringForce =  Calculus::rotateVector(n1/abs(n1) * 3.141592 / 45.0, facing);
	}

	//This is the desired direction, to be affected by walls and obstacles.
	return steeringForce * maxSpeed;
}

/******** Function to compute a steering force to avoid walls ********/
Vector3d Animat::avoidWalls(Wall* walls)
{
	Vector3d steeringForce = Vector3d::Zero();	//Initialize an homogeneous vector.

	for(int I=0; I<3; I++)                          //Check the three feelers.
	{
		int closestWall = -1;             //Keep track of closest wall to this feeler.
		Vector3d pClosestIntersection, pIntersection;	//Intersection points.
		double minimumDistance = 1000.0;          //Like a Max distance to start with.
		double closestWallDistance;               //Wall distances.

		for(int J=0; J<4; J++)                    //Traverse the four walls.
		{
			if(Calculus::intersectSegments(walls[J].getStartPoint(),
				walls[J].getSegment(), position, feelers[I] - position,
				pIntersection) == 1)
```

```
                {
                        Vector3d vDistance = pIntersection - position;  //Vector to
                                        //determine the distance of the intersection.
                        double wallDistance = vDistance.norm();
                        if(wallDistance < minimumDistance)       //Closest wall so far?
                        {
                                pClosestIntersection = pIntersection;    //Store closest
                                                                         //intersection point.
                                closestWall = J;
                                closestWallDistance = wallDistance;
                        }
                }
        }
        if(closestWall >= 0)                     //Is there an intersection?
        {
                Vector3d vOvershoot = feelers[I] - pClosestIntersection;
                //Add the steering force as vector.
                steeringForce += walls[closestWall].getNormal() * vOvershoot.norm();
        }
    }

    return steeringForce;       //Return the accumulation of forces over the 3 feelers and
                                //walls.
}

/* Function to perform obstacle avoidance given a list of neighboring obstacles */
/*** Information contained in obstacles must be already given in animat frame ***/
Vector3d Animat::obstacleAvoidance()
{
    double intersectionBoxLength = 1.0;
    Obstacle* closestObstacle = NULL;               //Pointer to the closest obstacle.
    double closestObstacleDistance = 1000.0;        //To determine the distance to the
                                                    //closest obstacle.
    vector <Obstacle*>::const_iterator obstacle = obstacles.begin();     //Set an
                                            //iterator to the vector of obstacles.

    while(obstacle < obstacles.end())       //Traverse the list of obstacles.
    {
        Vector3d localPosition = (*obstacle)->getLocalPosition();
        if(localPosition[0] >= 0.0)         //Consider only obstacles that are in front
                                            //of the animat.
        {
            double expandedRadius = (*obstacle)->getBoundingRadius()
                + personalRadius*Calculus::getMainScale();

            if(fabs(localPosition[1]) < expandedRadius)     //Check for possible
                                                            //intersections.
            {
                double sqrtPart = sqrt(expandedRadius*expandedRadius
                    - localPosition[1]*localPosition[1]);    //Intersect
                                                             //circle with x axis.
                double intersection = localPosition[0] - sqrtPart;
                if(intersection < 0.0)            //Take just the positive part.
                    intersection = localPosition[0] + sqrtPart;

                //Verify if this intersection is the closest so far.
                if(intersection < closestObstacleDistance
                    && intersection < intersectionBoxLength)
                {
                    closestObstacleDistance = intersection;  //Store the
                                //intersection point for later force computation.
```

```
                                closestObstacle = *obstacle;      //Store the pointer to
                                                                  //the closest obstacle.
                        }
                }
        }
        obstacle++;          //Move iterator.
    }

    Vector3d steeringForce = Vector3d::Zero();      //Steering force to return.
    if(closestObstacle != NULL)                     //Was there a closest obstacle?
    {
        steeringForce[0] = closestObstacleDistance;
        steeringForce = steeringForce - closestObstacle->getLocalPosition();
        steeringForce[2] = 0.0;                     //Get normal to intersection point.
        if((float)steeringForce[0] != 0.0 && (float)steeringForce[1] != 0.0)
                                //Verify if it is possible to get a normalized vector.
        {
            steeringForce.normalize();
            steeringForce *= closestObstacle->getBoundingRadius(); //Scale force to
                                                                  //radius of obstacle.
            steeringForce[1] *= 1.0 + (intersectionBoxLength
                - steeringForce[0])/intersectionBoxLength;      //Increase
                                                //lateral force with closeness.
            return Calculus::animatToWorldFrame(position, facing,
                steeringForce*closestObstacle->getMultiplier());//Return
                                        //steering force in world coordinates.
        }
        else
            return Vector3d::Zero();
    }
    else
        return Vector3d::Zero();

}

/** Function to compute the total steering force for getting a new **/
/****************** velocity and facing vectors *********************/
Vector3d Animat::computeSteeringForce(Wall* walls)
{
    Vector3d totalForce = Vector3d::Zero();  //Initialize the accumulative force vector.
    Vector3d force = Vector3d::Zero();       //Individual force.

    //Compute individual forces in prioritized order.
    //To avoid unnecessary comparisons, test walls only when position is close to walls.
    if(10.0 - fabs(position[0]) < viewRadius/2.0
        || 10.0 - fabs(position[1]) < viewRadius/2.0)
    {
        force = avoidWalls(walls) * 0.8;         //Multiply it by a priority weight.
        if(!accumulateForce(totalForce, force))
        {
            return totalForce;                   //Cannot continue adding forces.
        }
    }

    force = obstacleAvoidance() * 0.8;
    if(!accumulateForce(totalForce, force))
    {
        return totalForce;                       //Cannot continue adding forces.
    }

    if(brain.getInputSum() > 0.0 && brain.getOutputSum() > 0.0
```

```
		&& Calculus::randomGenerator(0.0, 1.0) < 0.98)	//Is there a significant
								//change in the input over
								//time?
	{
		force = neuralSteeringForce * 0.5;
		if(!accumulateForce(totalForce, force))
		{
			return totalForce;
		}
	}
	else	//Use a random steering force to compensate for a missing decision instead.
	{
		force = randomWalk() * 0.5;
		if(!accumulateForce(totalForce, force))
		{
			return totalForce;
		}
	}

	return totalForce;
}

/************ Function to accumulate the steering forces *************/
bool Animat::accumulateForce(Vector3d& totalForce, const Vector3d& forceToAdd)
{
	double magnitudeSoFar = totalForce.norm();		//Get the remainder space for
								//additional forces.
	double magnitudeRemaining = maxSpeed - magnitudeSoFar; //Maxspeed is the maximum
								//moving-forward achievable by the animal.

	if(magnitudeRemaining <= 0.0)		//No more space?
		return false;

	double magnitudeToAdd = forceToAdd.norm();		//Magnitude of the incoming force.

	if(magnitudeToAdd < magnitudeRemaining)		//Able to add all of the new force?
		totalForce += forceToAdd;
	else		//Otherwise, truncate the incoming force and add it.
		totalForce += forceToAdd.normalized() * magnitudeRemaining;

	return true;		//totalForce increased successfully.
}

/********** Function to assign the new velocity and facing **********/
/************ steeringForce must be in animat coordinates ************/
Vector3d Animat::computeNewVelocityAndFacing(const Vector3d& steeringForce, double
worldSteeringForceMagnitude)
{
	Vector3d newVelocity = Vector3d::Zero();		//Initialize new velocity.
	double angle = atan2(steeringForce[1], steeringForce[0]);	//Get the angle.
	if(angle > 3.141592)
		angle = 3.141592 - angle;		//Get negative and positive angles.

	if(fabs(angle) > 3.141592/45.0)		//Angle greater than 4 degrees?
	{
		angle = (angle/fabs(angle)) * 3.141592/45.0;	//Truncate it.
	}

	newVelocity = Calculus::rotateVector(angle, facing);	//Rotate facing (given in
								//world coordinates).
	facing = newVelocity.normalized();
```

```
        velocity = facing * worldSteeringForceMagnitude;        //Here we keep the steering
                                                                //force magnitude in world
                                                                //coordinates.

        speed = worldSteeringForceMagnitude;                    //New speed (which will never
                                                                //be greater than maxSpeed).

        return velocity;            //Return new velocity to world.
}
```

A.3. Brain Class

The following C++ code contains the most relevant function definitions for the Brain class.

```
/****************** Initialize some constants ******************/
const double Brain::theta1 = 0.022222;
const double Brain::theta2 = 0.02;

/************************ Constructor **************************/
Brain::Brain()
{
        //Maximally responding signals.
        maxArgGoal = 0;
        maxArgOutput = 0;
        maxArgInput = 0;

        //Tester for input change.
        inputSum = 0.0;

        //Tester for output change.
        outputSum = 0.0;

        //Initialize the neural network weights with random values.
        for(int I=0; I<totalGoals; I++)
        {
                raw_goals[I] = goals[I] = goals_1[I] = 0.0;     //Initially the animat does
                                                        //not have any internal goal in mind.
                for(int J=0; J<totalOutputs; J++)
                {
                        output[J] = 0.0;                //Initialize outputs.
                        output_1[J] = 0.0;
                        for(int K=0; K<totalInputs; K++)  //Weights set randomly.
                                policyNetwork[K][J][I] = Calculus::randomGenerator(0.494, 0.506);
                }
        }

        //Initialize the value network weights.
        for(int I=0; I<totalInputs; I++)
                for(int J=0; J<totalGoals; J++)
                        valueNetwork[I][J] = 0.0;       //No associations.

        //Initialize the input and reinforcement signals.
        for(int I=0; I<totalInputs; I++)
                input[I] = input_1[I] = reinforcement[I] = input_angle[I]
                        = input_closeness[I] = 0.0;

        //Initialize parenting signals.
        parenting = parenting_1 = 0;
}
```

```
/** Function to copy the external input into the brain state ***/
void Brain::setInput(double in[], double angle[], double closeness[])
{
	inputSum = 0.0;		//Restart the tester for change in input.
	maxArgInput = 0;
	for(int I=0; I<totalInputs; I++)	//Copy last t-value into t-1.
	{
		//Then, calculate the new "brain" input.
		input_1[I] = input[I];
		input[I] = in[I];

		//Store the angle with the object.
		input_angle[I] = angle[I];

		//Store closeness value for each input object.
		input_closeness[I] = closeness[I];

		// Find at the same time the maximally responding input (closeness) //
		if(input_closeness[I] > input_closeness[maxArgInput])
			maxArgInput = I;
		//////////////////////////////////////////////////////////

		inputSum += abs(input[I]) + abs(input_1[I]);
	}
}

/** Function to copy the internal goals into the brain state **/
void Brain::setGoals(double g[])
{
	maxArgGoal = 0;

	for(int I=0; I<totalGoals; I++)		//Copy last t-value into t-1.
	{
		//Store memory for raw goals.
		raw_goals_1[I] = raw_goals[I];	//Raw goals can go from -inf to +inf.
		raw_goals[I] = g[I];

		//Generate the excitatory/inhibitory signal.
		unsigned int signal;
		switch(I)
		{
		case Hunger: signal = 1; break;		//Keep hunger always as is.
		case Libido: signal = !(sex || parenting); break;//not(sex or parenting).
		case Brood: signal = parenting && (!sex); break;//parenting and not sex.
		}

		//Filter according to current value in parenting node.
		goals_1[I] = goals[I];
		goals[I] = raw_goals[I] * (double)signal;

		//Check limits, because although raw_goals may be [-inf, +inf], no the
		//filtered goals.
		if(goals[I] < 0.0) goals[I] = 0.0;
		else if(goals[I] > 1.0) goals[I] = 1.0;

		// Find at the same time the maximally responding goal //
		if(goals[I] > goals[maxArgGoal])
			maxArgGoal = I;
		//////////////////////////////////////////////////////////
	}
}
```

```
/** Function to perform decision making based on the values ***/
/* of the weights in the policy network and the current input */
void Brain::makeDecision(double o[])
{
	//At this point, inputs must be already set (or if 0, perform random walk).
	outputSum = 0.0;		//For normalization purposes.

	for(int I=0; I<totalOutputs; I++)	//Collect values for each output.
	{
		output_1[I] = output[I];	//Save old output.

		output[I] = 0.0;		//Initialize the output.
		o[I] = 0.0;			//in both arrays (in brain and animat).
		for(int J=0; J<totalInputs; J++)	//Go over all inputs.
			for(int K=0; K<totalGoals; K++)	//Go over all internal goal sensors.
				output[I] += abs(input[J]) * goals[K] * policyNetwork[J][I][K];

		if(output[I] < 0.0)
			output[I] = 0.0;	//Truncate values out of [0,1].
		else
		{
			if(output[I] > 1.0)
				output[I] = 1.0;
		}

		outputSum += output[I];	//Accumulate output.
	}
	if(outputSum > 0.0)		//Was there any output activation?
	{
		maxArgOutput = 0;
		for(int I=0; I<totalOutputs; I++)
		{
			o[I] = output[I];

			// Find at the same time the maximally responding output //
			if(output[I] > output[maxArgOutput])
				maxArgOutput = I;
			//////////////////////////////////////////////////////////
		}
	}
}

/*** Function to update the policy weights during learning ***/
void Brain::adjustPolicyNetwork()
{

	//maxArgGoal and maxArgOutput must be set already!

	//Update weights for all input with a connection with maxArgOutput and maxArgGoal.
	for(int I=0; I<totalInputs; I++)
	{
		if(fabs(input_1[I]-input[I]) < theta1 && reinforcement[I] != 0.0)//Valid for
								//learning?
		{
			policyNetwork[I][maxArgOutput][maxArgGoal] +=
				reinforcement[I]*output[maxArgOutput];

			if(policyNetwork[I][maxArgOutput][maxArgGoal] < 0.0)	//Check
								//boundaries [0 ,1]
				policyNetwork[I][maxArgOutput][maxArgGoal] = 0.0;
```

```
                        else
                        {
                                if(policyNetwork[I][maxArgOutput][maxArgGoal] > 1.0)
                                        policyNetwork[I][maxArgOutput][maxArgGoal] = 1.0;
                        }
                }
        }
}

/* Function to compute the reinforcement signal for the policy network to use it */
void Brain::computeReinforcementSignals()
{
        //maxArgGoal must be set at this time already.

        for(int I=0; I<totalInputs; I++)
        {
                if(input_angle[I] < 0.05236)//A threshold for avoiding negative reinforcement
                                            //when it is approaching from front.
                {
                        reinforcement[I] = 0.0;
                }
                else
                {
                        reinforcement[I] = (input[I] - input_1[I])
                                * valueNetwork[I][maxArgGoal] * 10.0;     //Normal update.
                }
        }
}

/********** Function to adjust the weights in the value network ***********/
void Brain::adjustValueNetwork()
{
        //maxArgInput must be already set at this time.

        for(int I=0; I<totalGoals; I++)    //Assign credit to the goal that changed the most.
        {
                if(fabs(raw_goals_1[I]-raw_goals[I]) > theta2 && parenting <= parenting_1)
                {       //Using the unfiltered goal values.

                        valueNetwork[maxArgInput][I] += (raw_goals_1[I] - raw_goals[I])
                                * abs(input[maxArgInput]);

                        if(valueNetwork[maxArgInput][I] < -1.0)   //Cap values to be in
                                                                  //[-1, +1].
                                valueNetwork[maxArgInput][I] = -1.0;
                        else
                        {
                                if(valueNetwork[maxArgInput][I] > 1.0)
                                        valueNetwork[maxArgInput][I] = 1.0;
                        }
                }
        }
}

/************* Function to set the parenting value *************/
void Brain::setParenting(unsigned int val)
{
        parenting_1 = parenting;            //Keeps in memory last parenting value.
        parenting = val;                    //New parenting value.
}
```

A.4. Simulate Function

The "simulate" function updates the animats' state and the environment dynamics. The following C++ code defines how the Antarctica ecosystem operates at the implementation level.

```
/***************** Function to simulate ********************/
void Antartica::simulate(double time)
{
	//Set a maximum simulation time.
	if(time > 500.0)
		return;

	Vector3d steeringForce = Vector3d::Zero();
	Vector3d localSteeringForce = Vector3d::Zero();
	Vector3d oldPosition = Vector3d::Ones();
	Vector3i discreteOldPosition = Vector3i::Ones();
	Vector3i discreteNewPosition = Vector3i::Ones();
	Vector3d oldFacing = Vector3d::Zero();
	Vector3d newVelocity = Vector3d::Zero();
	Vector3d wFeelers[3];
	AnimatPtrList* ptrAnimatPtrList = NULL;
	AnimatPtrNode* animatNode = animatList.getStartAnimatPtrNode();	//Node that
	                                  //points to the beginning of the list of animats.

	//// Scoring variables ////
	float maleFood = 0;
	float malePoison = 0;
	float laidEggs = 0;
	float rejections = 0;
	float femaleFood = 0;
	float femalePoison = 0;
	float gifts = 0;
	int malesAlive = 0;		//Refers to tracked population.
	int totalMales = 0;		//Refers to whole population.
	int femalesAlive = 0;
	int totalFemales = 0;

	///// Update gifts that might have disappeared when some animat picked them up //////
	for(int I=0; I<giftCount; I++)
		giftList[I].updateGiftTicks();
	////////////////////////////////////////////////////////////////////////////////////

	//////////////////////////// Make the environment dynamic ////////////////////////////
	if(fmod((float)time, 50.0f) == 0.0f)
		shiftEnvironment();
	////////////////////////////////////////////////////////////////////////////////////

	while(animatNode != NULL)	//Iterate over all living animats.
	{
		Animat* animat = animatNode->getAnimatPtr();//Get the pointer to the actual
                                                     //animat in the list.

		// Recover old position and facing.
		oldPosition = animat->getPosition();
		oldFacing = animat->getFacing();
		discreteOldPosition = Calculus::worldToDiscreteFrame(oldPosition);	//Discrete
                                                                  //position of animat.

		//////////////////////// Implement costs of living ////////////////////////////
```

```
ptrAnimatPtrList = cells[discreteOldPosition[1]*nBinsPerSide +
       discreteOldPosition[0]].getAnimatPtrList();
animat->gettingOld(0.01);
animat->updateHunger();     //Update goal state and reduce timeouts.
animat->updatePoison();
animat->updateLibido();
animat->shiftCarryingGift();
animat->updateFemaleSensing();     //Males update timeouts for sensing females.
animat->updateNestSensing();
animat->updateGiftSensing();
animat->updateGrowingEggTime();    //Applies only to females.

if(animat->getHunger() >= 1.0)     //Animat died? (This reading is from goals
                            //that are going to be sent to brain at the end).
{
       ptrAnimatPtrList->removeAnimat(animat);   //Remove animat ptr from
                                                 //previous cell.

       animatNode = animatList.removeAnimatPtrNode(animatNode);//Delete animat
                     //from the world and keep track of the next in list.
       delete(animat);             //Destroy this animat object.
}
else
{
       // Recover vector of obstacles for their manipulation.
       vector <Obstacle*>* obstacles = animat->getObstacles();

       ///////// Update the animat's feelers into world coordinates /////////
       for(int J=0; J<3; J++)
       {
              wFeelers[J] = Calculus::animatToWorldFrame(oldPosition,
                     oldFacing, allFeelers[J]);
              wFeelers[J] *= mainScale;   //Stretch feelers to main scale.
       }
       animat->setFeelers(wFeelers);      //Copy feelers in world coordinates
                                          //into the animat.

       //////////////////// Computing the steering forces ////////////////////
       if(!animat->didSheReceiveGift())   //Avoid moving a female that has been
                                          //marked to get a food-gift.
              animat->makeDecision();     //Policy network determines new
                                          //action.
       bool doneThisTimeStep = !animat->canDoSomethingNextTimeStep();
              //Indicates that male animat can do one thing per time step.
       animat->updateParenting();         //Do this after determining if animat
                            //is allowed to perform an action this time step.
       if(animat->rewardGiftThisTimeStep())
       {
              animat->endReproduction();  //This is the only action the animat
                                          //can do to complete reproduction.
              doneThisTimeStep = true;
       }

       if(!animat->didSheReceiveGift())   //Avoid moving a female that has been
                     //marked to get a good gift. (No new facing vector).
       {
              steeringForce = animat->computeSteeringForce(walls);   //The
              //obstacles vector must have obstacles from last time step.
              localSteeringForce = Calculus::worldToAnimatFrame(oldPosition,
                      oldFacing, steeringForce); //Necessary to compute +/-
                                                 //angle.
```

```
        newVelocity = animat->computeNewVelocityAndFacing(
                localSteeringForce, steeringForce.norm());
}

for(unsigned J=0; J<obstacles->size(); J++)//Destroy dynamic obstacles
                                            //collected in last sensing.
{
        delete obstacles->at(J);
}
obstacles->clear();                         //Empty the vector container.

///////////// Move the animat according to new velocity ///////////////
Vector3d newPosition = oldPosition;
if(!animat->didSheReceiveGift())            //Avoid female to move to
                                    //register who gave her a food gift.
        newPosition = oldPosition + newVelocity * 0.01;

if(newPosition[0] < -10.0)          //Validate new position.
        newPosition[0] = -9.5;
else
{
        if(newPosition[0] >= 10.0)
                newPosition[0] = 9.5;
}
if(newPosition[1] > 10.0)
        newPosition[1] = 9.5;
else
{
        if(newPosition[1] <= -10.0)
                newPosition[1] = -9.5;
}
animat->setPosition(newPosition);  //Move the animat.
discreteNewPosition =
        Calculus::worldToDiscreteFrame(animat->getPosition());

if(discreteOldPosition != discreteNewPosition)  //Did it move from one
                                                //cell to another?
{
        if(ptrAnimatPtrList->removeAnimat(animat))      //Remove animat
                                            //ptr from previous cell.
        {
                ptrAnimatPtrList = cells[discreteNewPosition[1]*
                        nBinsPerSide + discreteNewPosition[0]]
                                .getAnimatPtrList();
                ptrAnimatPtrList->insertAnimat(animat);  //Insert animat
                                                //ptr into new cell.
        }
}

///// Get the current neighbors (food sources, other animats, etc) /////
double* rawInput = animat->getInput();    //Store here any sensing,
                //which will be processed as input for a NN in the animat.
double* rawInputAngle = animat->getAngleInput();//Store the angles.
double* rawInputCloseness = animat->getClosenessInput();//Store the
                //closeness of an object.
vector <int> neighborCells;         //Dynamic array for storing the
                                    //indices of neighboring cells.
Calculus::getNeighborCellIndices(discreteOldPosition[0],
        discreteOldPosition[1], &neighborCells);
bool pickedUpGift = false;          //Indicates if animat picked up a
                                    //food-gift this time step.
```

```
FoodSource* closestFoodLeft = NULL;      //Closest food sources.
FoodSource* closestFoodRight = NULL;
double FdL = 1000.0; //Closest food source distances in both directions.
double FdR = 1000.0;
double FaL = 4.0;    //Closest food source angles in both directions.
double FaR = 4.0;
double FcL = 0.0;    //Closeness measure.
double FcR = 0.0;

FoodSource* closestPoisonLeft = NULL;//Closest poison units.
FoodSource* closestPoisonRight = NULL;
double PdL = 1000.0; //Closest poisonous distances in both directions.
double PdR = 1000.0;
double PaL = 4.0;    //Closest poison angles in both directions.
double PaR = 4.0;
double PcL = 0.0;    //Closeness measure.
double PcR = 0.0;

Nest* closestWarmLeft = NULL;      //Closest warm nest.
Nest* closestWarmRight = NULL;
double WdL = 1000.0; //Closest warm nest distances in both directions.
double WdR = 1000.0;
double WaL = 4.0;    //Closest warm nest angles in both directions.
double WaR = 4.0;
double WcL = 0.0;    //Closeness measure.
double WcR = 0.0;

Nest* closestFrozenLeft = NULL;    //Closest frozen nest.
Nest* closestFrozenRight = NULL;
double CdL = 1000.0; //Closest frozen nest distances in both directions.
double CdR = 1000.0;
double CaL = 4.0;    //Closest frozen nest angles in both directions.
double CaR = 4.0;
double CcL = 0.0;    //Closeness measure.
double CcR = 0.0;

Animat* closestFemaleGiftLeft = NULL;//Closest female when he has a gift
                                     //ptrs.
Animat* closestFemaleGiftRight = NULL;
double AdL = 1000.0;        //Closest female animat distances in both
                            //directions.
double AdR = 1000.0;
double AaL = 4.0;    //Closest female animat angles in both directions.
double AaR = 4.0;
double AcL = 0.0;    //Closeness measure.
double AcR = 0.0;

Animat* closestFemaleNoGiftLeft = NULL;//Closest female when he does not
                                       //have a gift ptrs.
Animat* closestFemaleNoGiftRight = NULL;
double UdL = 1000.0;        //Closest female animat distances in both
                            //directions.
double UdR = 1000.0;
double UaL = 4.0;    //Closest female animat angles in both directions.
double UaR = 4.0;
double UcL = 0.0;    //Closeness measure.
double UcR = 0.0;

Gift* closestGiftLeft = NULL;      //Closest gift ptrs.
Gift* closestGiftRight = NULL;
```

```
double GdL = 1000.0;        //Closest gift distances in both directions.
double GdR = 1000.0;
double GaL = 4.0;           //Closest gift angles in both directions.
double GaR = 4.0;
double GcL = 0.0;           //Closeness measure.
double GcR = 0.0;

Animat* closestMaleLeft = NULL;    //Closest male with gift ptrs.
Animat* closestMaleRight = NULL;
double MdL = 1000.0;        //Closest male distances in both directions.
double MdR = 1000.0;
double MaL = 4.0;           //Closest male angles in both directions.
double MaR = 4.0;
double McL = 0.0;           //Closeness measure.
double McR = 0.0;

for(unsigned J = 0; J < neighborCells.size(); J++)//Traverse all of the
                                          //neighbor cell indices.
{
       ///////////////// Working with food sources ////////////////////
       if(cells[neighborCells[J]].getFoodSourceId() != -1)    //Is there
                                          //a food source in this cell.
       {
              FoodSource* ptrFs=&foodSourceList[cells[neighborCells[J]]
                     .getFoodSourceId()];
              Vector3d vDistance = ptrFs->getPosition()
                     - animat->getPosition();    //Distance vector
                                   //between animat and food source.
              double distance = vDistance.norm();//Distance has not to
                                   //be 0 for this to work!

              double temperatureDistance = distance
                     - animat->getTemperatureRadius();  //To check if
                                          //affected by temperature.
              if(temperatureDistance - ptrFs->getConsumptionRadius()
                     < 0.0 && !ptrFs->isPoisonous()
                     && ptrFs->isAvailable())
              {
                     temperatureDistance = abs(temperatureDistance);
                     double alpha = temperatureDistance /
                            (animat->getTemperatureRadius()
                            + ptrFs->getConsumptionRadius());
                     ptrFs->updateTemperature(animat->getTemperature()
                            * alpha);//Update food source's temperature.
              }

              if(distance - ptrFs->getConsumptionRadius()
                     < animat->getViewRadius())  //In range of view?
              {
                     Vector3d localPosition = Calculus::
                            worldToAnimatFrame(animat->getPosition(),
                            animat->getFacing(), ptrFs->getPosition());

                     if(localPosition[0] > 0.0)  //Can animat see the
                                                 //core of food source?
                     {
                            Vector3d vDistanceNormalized = vDistance
                                   .normalized();//For computing the
                                   //angle with the animat's facing.
                            double dotProd = vDistanceNormalized.dot
                                   (animat->getFacing());
```

```
                                        dotProd = (dotProd < -1.0)? -1.0: dotProd;
                                        //Verify that angles will be valid.
                                        dotProd = (dotProd > 1.0)? 1.0: dotProd;
                                        double angle = acos(dotProd);
                                        //Compute the angle.

////////////////////////// Working with food sources only ////////////////////
if(!ptrFs->isPoisonous() && ptrFs->isAvailable())      //Is it an edible food source?
{
        //////////////////////// Create an obstacle object ////////////////////
        Obstacle* obs = new Obstacle(ptrFs->getCollisionRadius()*mainScale,
                localPosition, 2.0 + 4.0*animat->getSpeed()/animat->getMaxSpeed());
        obstacles->push_back(obs);  //Attach dynamically-created object at the end.

        ////////////////////// Get the closeness measure ////////////////////
        double thisClose = closeness(newPosition, newVelocity, animat->getSpeed(),
                 angle, distance,
        distance + ptrFs->getConsumptionRadius(), ptrFs->getPosition());

        if(localPosition[1] > 0.0)                     //Is food source to the left?
        {
                if(thisClose > FcL)        //Find the closest food source.
                {
                        closestFoodLeft = ptrFs;          //Store food source ptr.
                        FdL = distance;                   //Distance
                        FaL = angle;                      //and angle.
                        FcL = thisClose;
                }
        }
        else
        {
                if(thisClose > FcR)        //Find the closest food source.
                {
                        closestFoodRight = ptrFs;         //Store food source ptr.
                        FdR = distance;                   //Distance
                        FaR = angle;                      //and angle.
                        FcR = thisClose;
                }
        }

        ////// Feeding from the first closest food source it detects ////////
        if(!doneThisTimeStep && animat->canSenseFood())
        {
                if(distance - ptrFs->getConsumptionRadius()
                        < animat->getPersonalRadius())    //Has the animat introduced
                                                          //its body in the food source?
                {
                        if(angle <= 1.0472)  //Is it aligned between +/-60 degrees?
                        {
                                animat->feed();           //Replenish resources.
                                if(FcL > FcR)        //Give the highest closeness to
                                                     //closest object.
                                        FcL = 1.0;
                                else
                                        FcR = 1.0;
                                doneThisTimeStep = true;  //Animat cannot do something
                                                     //else this time step.
                        }
                }
        }
}
```

```
        ///////////////// Working with poisonous food sources only ////////////////
        if(ptrFs->isPoisonous())
        {
                ////////////////////// Create an obstacle object ////////////////////
                Obstacle* obs = new Obstacle(ptrFs->getCollisionRadius()*mainScale,
                        localPosition, 2.0 + 4.0*animat->getSpeed()/animat->getMaxSpeed());
                obstacles->push_back(obs);  //Attach dynamically-created object at the end.

                ////////////////////// Get the closeness measure ////////////////////
                double thisClose = closeness(newPosition, newVelocity, animat->getSpeed(),
                        angle, distance,
                distance + ptrFs->getConsumptionRadius(), ptrFs->getPosition());

                if(localPosition[1] > 0.0)        //Is poisonous food source to the left?
                {
                        if(thisClose > PcL)       //Find the closest poisonous food source.
                        {
                                closestPoisonLeft = ptrFs;  //Store poisonous food source ptr.
                                PdL = distance;             //Distance
                                PaL = angle;                //and angle.
                                PcL = thisClose;
                        }
                }
                else
                {
                        if(thisClose > PcR)       //Find the closest poisonous food source.
                        {
                                closestPoisonRight = ptrFs; //Store food poisonous source ptr.
                                PdR = distance;             //Distance
                                PaR = angle;                //and angle.
                                PcR = thisClose;
                        }
                }

                ////// Getting hurt by the poisonous food source ////////
                if(!doneThisTimeStep && animat->canSensePoison())
                {
                        if(distance - ptrFs->getConsumptionRadius()
                                < animat->getPersonalRadius())     //Has the animat touched the
                                                                   //poisonous food source?
                        {
                                if(angle <= 1.0472)        //Is it aligned between +/- degrees?
                                {
                                        animat->hurt(Brain::Poison);//Hurt animat for learning to
                                                                    //happen (with poison).
                                        if(PcL > PcR)       //Give the highest closeness to
                                                            //closest object.
                                                PcL = 1.0;
                                        else
                                                PcR = 1.0;
                                        doneThisTimeStep = true;    //Animat cannot do something
                                                                    //else this time step.
                                }
                        }
                }
        }
}
```

```
                }
        }
```

```
////////////////////////// Working with nests ////////////////////////////
if(cells[neighborCells[J]].getNestId() != -1)   //Is there a nest in this cell.
{
        Nest* ptrN = &nestList[cells[neighborCells[J]].getNestId()];
        Vector3d vDistance = ptrN->getPosition() - animat->getPosition();    //Distance
                                                //vector between animat and nest.
        double distance = vDistance.norm();     //Distance has not to be 0 for this
                                                //to work!

        if(distance - ptrN->getDetectionRadius() < animat->getViewRadius())  //In range
                                                                              //of view?
        {
                Vector3d localPosition = Calculus::worldToAnimatFrame(
                        animat->getPosition(), animat->getFacing(), ptrN->getPosition());

                if(localPosition[0] > 0.0)  //Can animat see a minimum part of nest?
                {
                        Vector3d vDistanceNormalized = vDistance.normalized(); //For
                                        //computing the angle with the animat's facing.
                        double dotProd = vDistanceNormalized.dot(animat->getFacing());
                        dotProd = (dotProd < -1.0)? -1.0: dotProd;//Verify that angles
                                                        //will be valid.
                        dotProd = (dotProd > 1.0)? 1.0: dotProd;
                        double angle = acos(dotProd);           //Compute the angle.

                        //////////////////// Create an obstacle object /////////////////
                        Obstacle* obs = new Obstacle(
                                ptrN->getCollisionRadius()*mainScale, localPosition,
                                2.0 + 4.0*animat->getSpeed()/animat->getMaxSpeed());
                        obstacles->push_back(obs);  //Attach dynamically-created object
                                                    //at the end.

                        ////////////////// Get the closeness measure /////////////////
                        double thisClose = closeness(newPosition, newVelocity,
                                animat->getSpeed(), angle, distance,
                                distance + ptrN->getDetectionRadius(),
                                ptrN->getPosition());

                        ////////////////////// Working with warm nests ////////////////////
                        if(!ptrN->isFrozen())
                        {
                                if(localPosition[1] > 0.0)  //Is nest to the left?
                                {
                                        if(thisClose > WcL)  //Find the closest nest.
                                        {
                                                closestWarmLeft = ptrN;//Store nest ptr.
                                                WdL = distance;      //Distance
                                                WaL = angle;         //and angle.
                                                WcL = thisClose;
                                        }
                                }
                                else
                                {
                                        if(thisClose > WcR)  //Find the closest nest.
                                        {
                                                closestWarmRight = ptrN;//Store nest ptr.
                                                WdR = distance;      //Distance
                                                WaR = angle;         //and angle.
                                                WcR = thisClose;
                                        }
                                }
```

```
                ////// Bumping with the closest nest ////////
                if(!doneThisTimeStep && animat->canSenseWarmNests())//Has
                        //the animat introduced its body in the nest?
                {
                        if(distance - ptrN->getDetectionRadius()
                                < animat->getPersonalRadius())//Is it at a
                                                        //reasonable distance?
                        {
                                if(angle <= 1.0472)  //Is it aligned between
                                                        //+/- 60 degrees?
                                {
                                        if(animat->bumpNest(ptrN))  //Did
                                                        //something with nest.
                                        {
                                                if(animat->droppedEgg())//Did
                                                //it lay an egg on the nest?
                                                {
                                                        //Create a new anima
                                                        giveBirth(animat);
                                                        //Leave evidence of
                                                        //birth success.
                                                        ptrN->addWhiteEgg();
                                                }
                                                if(WcL > WcR) //Give the
                                                //highest closeness to closest
                                                //object.
                                                        WcL = 1.0;
                                                else
                                                        WcR = 1.0;

                                                doneThisTimeStep = true;
                                                //Animat cannot do something
                                                //else this time step.
                                        }
                                }
                        }
                }
        }
        else
        {       ////////////// Working with frozen nests ////////////////
                if(localPosition[1] > 0.0)  //Is nest to the left?
                {
                        if(thisClose > CcL)  //Find the closest nest.
                        {
                                closestFrozenLeft = ptrN;//Store nest ptr.
                                CdL = distance;      //Distance
                                CaL = angle;         //and angle.
                                CcL = thisClose;
                        }
                }
                else
                {
                        if(thisClose > CcR)  //Find the closest nest.
                        {
                                closestFrozenRight = ptrN;//Store nest ptr.
                                CdR = distance;      //Distance
                                CaR = angle;         //and angle.
                                CcR = thisClose;
                        }
                }
```

```
                                ////// Bumping with the closest nest ////////
                                if(!doneThisTimeStep && animat->canSenseFrozenNests())
                                //Has the animat introduced its body in the nest?
                                {
                                        if(distance - ptrN->getDetectionRadius()
                                                < animat->getPersonalRadius())//Is it at a
                                                                        //reasonable distance?
                                        {
                                                if(angle <= 1.0472)  //Is it aligned between
                                                                        //+/- 60 degrees?
                                                {
                                                        if(animat->bumpNest(ptrN))  //Did
                                                                        //something with nest.
                                                        {
                                                                if(animat->droppedEgg())//Did
                                                                //it lay an egg on the nest?
                                                                {
                                                                        //Destroy the egg
                                                                        //because it was
                                                                        //dropped in a frozen
                                                                        //nest.
                                                                        killEgg(animat);

                                                                        //Leave evidence of
                                                                        //happening.
                                                                        ptrN->addBlackEgg();
                                                                }
                                                                if(CcL > CcR) //Give the
                                                                //highest closeness to closest
                                                                //object.
                                                                        CcL = 1.0;
                                                                else
                                                                        CcR = 1.0;
                                                                //Animat cannot do something
                                                                //else this time step.
                                                                doneThisTimeStep = true;
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }
}

////////////////////////// Working with food-gifts ////////////////////////////
if(cells[neighborCells[J]].getGiftId() != -1 && giftList[cells[neighborCells[J]]
        .getGiftId()].isAvailable())      //Is there an available gift in this cell.
{
        Gift* ptrR = &giftList[cells[neighborCells[J]].getGiftId()];
        Vector3d vDistance = ptrR->getPosition() - animat->getPosition();//Distance
                                                //vector between animat and gift.
        double distance = vDistance.norm();     //Distance has not to be 0 for this
                                                //to work!

        if(distance - ptrR->getDetectionRadius() < animat->getViewRadius()) //In range
                                                                             //of view?
        {
                Vector3d localPosition = Calculus::worldToAnimatFrame(
                        animat->getPosition(), animat->getFacing(), ptrR->getPosition());
```

```
if(localPosition[0] > 0.0)  //Can animat see a minimum part of gift?
{
        Vector3d vDistanceNormalized = vDistance.normalized(); //For
        //computing the angle with the animat's facing.
        double dotProd = vDistanceNormalized.dot(animat->getFacing());
        dotProd = (dotProd < -1.0)? -1.0: dotProd;//Verify that angles
                                                  //will be valid.
        dotProd = (dotProd > 1.0)? 1.0: dotProd;
        double angle = acos(dotProd);             //Compute the angle.

        //////////////////// Create an obstacle object ////////////////////
        Obstacle* obs = new Obstacle(ptrR->getCollisionRadius()
                *mainScale, localPosition,
                2.0 + 4.0*animat->getSpeed()/animat->getMaxSpeed());
        obstacles->push_back(obs);  //Attach dynamically-created object
                                    //at the end.

        //////////////////// Get the closeness measure ////////////////////
        double thisClose = closeness(newPosition, newVelocity,
                animat->getSpeed(), angle, distance,
                distance + ptrR->getDetectionRadius(),
                ptrR->getPosition());

        if(localPosition[1] > 0.0)  //Is food-gift to the left?
        {
                if(thisClose > GcL)  //Find the closest gift.
                {
                        closestGiftLeft = ptrR;//Store gift ptr.
                        GdL = distance;      //Distance
                        GaL = angle;         //and angle.
                        GcL = thisClose;
                }
        }
        else
        {
                if(thisClose > GcR)  //Find the closest gift.
                {
                        closestGiftRight = ptrR;    //Store gift ptr.
                        GdR = distance;             //Distance
                        GaR = angle;                //and angle.
                        GcR = thisClose;
                }
        }

        ////// Bumping with the closest gift ////////
        if(!doneThisTimeStep && animat->canSenseGifts())
        {
                if(distance - ptrR->getDetectionRadius()
                        < animat->getPersonalRadius())//Is it at a
                                                      //reasonable distance?
                {
                        if(angle <= 1.0472)  //Is it aligned between +/-60
                                             //degrees?
                        {
                                if(animat->bumpGift())//Did something with
                                                      //gift.
                                {
                                        pickedUpGift = true;
                                        ptrR->pickUp();//Make gift disappear
                                                      //for a while.
```

```
                                                        if(GcL > GcR) //Give the highest
                                                               //closeness to closest object.
                                                               GcL = 1.0;
                                                        else
                                                               GcR = 1.0;
                                                        doneThisTimeStep = true;//Animat
                                                        //cannot do something else this time
                                                        //step.
                                                }
                                        }
                                }
                        }
                }
        }
}

//////////////////// Working with other animats ////////////////////////
if(cells[neighborCells[J]].getAnimatPtrList()->getSize() > 0) //Are there animats on
                                                              //this cell?
{
        ptrAnimatPtrList = cells[neighborCells[J]].getAnimatPtrList();
        AnimatPtrNode* aNode = ptrAnimatPtrList->getStartAnimatPtrNode();

        while(aNode != NULL) //Check all of the animats in that cell.
        {
                Animat* cAnimat = aNode->getAnimatPtr();  //Get the actual animat ptr in
                                                          //that node.
                if(cAnimat->getId() != animat->getId())   //Skip itself.
                {
                        Vector3d vDistance = cAnimat->getPosition()
                                - animat->getPosition();//Distance vector between animats.
                        double distance = vDistance.norm();//Distance has not to be 0!

                        if(distance - cAnimat->getTemperatureRadius()
                                < animat->getViewRadius())  //In range of view?
                        {
                                Vector3d localPosition = Calculus::worldToAnimatFrame(
                                        animat->getPosition(),
                                        animat->getFacing(), cAnimat->getPosition());

                                if(localPosition[0] > 0.0)  //Can animat sense a minimum
                                                //part of other animat's temperature?
                                {
                                        ///////////// Create an obstacle object ///////////
                                        Obstacle* obs = new Obstacle(
                                                cAnimat->getPersonalRadius()*mainScale,
                                                localPosition, 2.0
                                                + cAnimat->getSpeed()
                                                /cAnimat->getMaxSpeed()*4.0);
                                        obstacles->push_back(obs);  //Attach dynamically-
                                                          //created object at the end.

                                        Vector3d vDistanceNormalized =
                                                vDistance.normalized();//For computing
                                                                       //angle.
                                        double dotProd = vDistanceNormalized.dot(
                                                animat->getFacing());
                                        dotProd = (dotProd < -1.0)? -1.0: dotProd;//Verify
                                                          //that angles will be valid.
                                        dotProd = (dotProd > 1.0)? 1.0: dotProd;
                                        double angle = acos(dotProd);//Compute the angle.
```

```
                        ////////// Get the closeness measure /////////////
                        double thisClose = closeness(newPosition,
                               newVelocity, animat->getSpeed(),
                               angle, distance, distance
                               + cAnimat->getTemperatureRadius(),
                               cAnimat->getPosition());

////////////////////// Male proposing a female ////////////////////
if(animat->getSex()==0 && cAnimat->isFemaleReady())
{
       if(animat->isCarryingGift())      //Allows an animat "seeing"
                               /females with two different perspectives.
       {
              /// Working with females when he has a gift to exchange //
              if(localPosition[1] > 0.0)  //Is female to the left?
              {
                     if(thisClose > AcL)  //Find the closest female.
                     {
                            closestFemaleGiftLeft = cAnimat;   //Store pointer
                                                               //to the animat.
                            AdL = distance;      //Distance
                            AaL = angle;         //and angle.
                            AcL = thisClose;
                     }
              }
              else
              {
                     if(thisClose > AcR)  //Find the closest female.
                     {
                            closestFemaleGiftRight = cAnimat;  //Store pointer
                                                               //to animat.
                            AdR = distance;      //Distance
                            AaR = angle;         //and angle.
                            AcR = thisClose;
                     }
              }
              /////// Propose to get an egg from this female ////////
              if(!doneThisTimeStep && animat->canSenseFemales())     //Check if
                                   //it is able to consult females.
              {
                     //Check if female is also looking at male.
                     Vector3d maleLocalPosition = Calculus::worldToAnimatFrame(
                            cAnimat->getPosition(), cAnimat->getFacing(),
                            animat->getPosition());
                     if(maleLocalPosition[0] > 0.0)
                     {
                            if(distance - cAnimat->getTemperatureRadius()
                                   < animat->getPersonalRadius())     //Allow
                                   //certain closeness between female and male.
                            {
                                   if(angle <= 1.0472)  //Is it aligned between
                                                        //+/- 60 degrees?
                                   {
                                          if(animat->reproduce(cAnimat))//Male
                                          //animat does it part if possible.
                                          {
                                                 if(AcL > AcR) //Give the
                                                 //highest closeness to closest
                                                 //object.
                                                        AcL = 1.0;
```

```
                                                        else
                                                               AcR = 1.0;
                                                        //Animat cannot do something
                                                        //else this time step.
                                                        doneThisTimeStep = true;
                                                }
                                        }
                                }
                        }
                }
        }
        else
        {
                /////// Working with females when he does not have a gift ///////
                if(localPosition[1] > 0.0)          //Is female to the left?
                {
                        if(thisClose > UcL)         //Find the closest female.
                        {
                                closestFemaleNoGiftLeft = cAnimat;       //Store
                                                        //pointer to the animat.
                                UdL = distance;     //Distance
                                UaL = angle;        //and angle.
                                UcL = thisClose;
                        }
                }
                else
                {
                        if(thisClose > UcR)  //Find the closest female.
                        {
                                closestFemaleNoGiftRight = cAnimat;//Store pointer
                                                                //to animat.
                                UdR = distance;     //Distance
                                UaR = angle;        //and angle.
                                UcR = thisClose;
                        }
                }

                /////// Propose to get an egg from this female ////////
                if(!doneThisTimeStep && animat->canSenseFemales())     //Check if
                                        //it is able to consult females.
                {
                        //Check if female is also looking at male.
                        Vector3d maleLocalPosition = Calculus::worldToAnimatFrame(
                        cAnimat->getPosition(), cAnimat->getFacing(),
                        animat->getPosition());
                        if(maleLocalPosition[0] > 0.0)
                        {
                                if(distance - cAnimat->getTemperatureRadius()
                                        < animat->getPersonalRadius())    //Allow
                                //certain closeness between female and male.
                                {
                                        if(angle <= 1.0472)//Is it aligned between
                                                                //+/- 60 degrees?
                                        {
                                                if(animat->reproduce(cAnimat))//If he
                                                //was carrying an egg, he will not be
                                                //affected.
                                                {
                                                        if(UcL > UcR) //Give the
                                                        //highest closeness to closest
                                                        //object.
```

```
                                                        UcL = 1.0;
                                                else
                                                        UcR = 1.0;
                                                doneThisTimeStep = true;
                                                //Animat cannot do something
                                                //else this time step.
                                        }
                                }
                        }
                }
        }
}
else
{
        /////////// Dealing with females that feel males with gifts ////////
        if(animat->getSex()==1 && cAnimat->isCarryingGift()
                && (animat->isFemaleReady() || animat->didSheReceiveGift()))
        //Enable male sensing if she has an egg or she does not have a pending
        //reward.
        {

                if(localPosition[1] > 0.0)  //Is male to the left?
                {
                        if(thisClose > McL)  //Find the closest male.
                        {
                                closestMaleLeft = cAnimat;  //Store pointer to the
                                                            //animat.
                                MdL = distance;             //Distance
                                MaL = angle;                //and angle.
                                McL = thisClose;
                        }
                }
                else
                {
                        if(thisClose > McR)  //Find the closest male.
                        {
                                closestMaleRight = cAnimat; //Store pointer to
                                                            //animat.
                                MdR = distance;             //Distance
                                MaR = angle;                //and angle.
                                McR = thisClose;
                        }
                }

                // Be rewarded if she was marked by giving her egg to male //
                if(!doneThisTimeStep && animat->didSheReceiveGift())
                {
                        if(distance - cAnimat->getTemperatureRadius()
                                < animat->getPersonalRadius())//Activation
                                                              //distance.
                        {
                                animat->rewardByGiftFromMale();    //Female always
                                        //executes her part after male proposes.
                                if(McL > McR) //Give the highest closeness to
                                              //closest object.
                                        McL = 1.0;
                                else
                                        McR = 1.0;
                                doneThisTimeStep = true;    //Animat cannot do
                                //something else this time step.
```

```
                                        }
                                }
                        }
                }
        }
}
```

```
                        }
                        aNode = aNode->getNext();
                }
        }
}

///////////////// Normalize neuron inputs in the animat ////////////////////
if(!animat->canSenseFood())                //Detect which units will be ignored.
        closestFoodLeft = closestFoodRight = NULL;
if(!animat->canSensePoison())
        closestPoisonLeft = closestPoisonRight = NULL;
if(animat->getSex()==1 || !animat->canSenseWarmNests())
        closestWarmLeft = closestWarmRight = NULL;
if(animat->getSex()==1 || !animat->canSenseFrozenNests())
        closestFrozenLeft = closestFrozenRight = NULL;
if(animat->getSex()==1 || !animat->canSenseFemales())
        closestFemaleGiftLeft = closestFemaleGiftRight = NULL;
if(animat->getSex()==1 || !animat->canSenseFemales())
        closestFemaleNoGiftLeft = closestFemaleNoGiftRight = NULL;
if(animat->getSex()==1 || !animat->canSenseGifts())
        closestGiftLeft = closestGiftRight = NULL;
if(animat->getSex()==0 )    //Males do not sense other males.
        closestMaleLeft = closestMaleRight = NULL;

Vector3d newFacing = animat->getFacing();              //Store the facing vector.

if(closestFoodLeft != NULL)
{
        rawInput[Brain::FdL] = activation(oldFacing, newPosition, newFacing,
                closestFoodLeft->getPosition(), FdL, animat->getViewRadius() +
                closestFoodLeft->getConsumptionRadius());
        rawInputAngle[Brain::FdL] = FaL;
        rawInputCloseness[Brain::FdL] = FcL;      //Closeness measure.
}
else
{
        rawInput[Brain::FdL] = 0.0;        //No activation at all.
        rawInputAngle[Brain::FdL] = 0.0;
        rawInputCloseness[Brain::FdL] = 0.0;
}

if(closestFoodRight != NULL)
{
        rawInput[Brain::FdR] = activation(oldFacing, newPosition, newFacing,
                closestFoodRight->getPosition(), FdR, animat->getViewRadius() +
                closestFoodRight->getConsumptionRadius());
        rawInputAngle[Brain::FdR] = FaR;
        rawInputCloseness[Brain::FdR] = FcR;      //Closeness measure.
}
else
{
        rawInput[Brain::FdR] = 0.0;        //No activation at all.
        rawInputAngle[Brain::FdR] = 0.0;
        rawInputCloseness[Brain::FdR] = 0.0;
```

```
}

if(closestPoisonLeft != NULL)
{
	rawInput[Brain::PdL] = activation(oldFacing, newPosition, newFacing,
		closestPoisonLeft->getPosition(), PdL, animat->getViewRadius() +
		closestPoisonLeft->getConsumptionRadius());
	rawInputAngle[Brain::PdL] = PaL;
	rawInputCloseness[Brain::PdL] = PcL;	//Closeness measure.
}
else
{
	rawInput[Brain::PdL] = 0.0;		//No activation at all.
	rawInputAngle[Brain::PdL] = 0.0;
	rawInputCloseness[Brain::PdL] = 0.0;
}

if(closestPoisonRight != NULL)
{
	rawInput[Brain::PdR] = activation(oldFacing, newPosition, newFacing,
		closestPoisonRight->getPosition(), PdR, animat->getViewRadius() +
		closestPoisonRight->getConsumptionRadius());
	rawInputAngle[Brain::PdR] = PaR;
	rawInputCloseness[Brain::PdR] = PcR;	//Closeness measure.
}
else
{
	rawInput[Brain::PdR] = 0.0;		//No activation at all.
	rawInputAngle[Brain::PdR] = 0.0;
	rawInputCloseness[Brain::PdR] = 0.0;
}

if(closestWarmLeft != NULL)
{
	rawInput[Brain::WdL] = activation(oldFacing, newPosition, newFacing,
		closestWarmLeft->getPosition(), WdL, animat->getViewRadius() +
		closestWarmLeft->getDetectionRadius());
	rawInputAngle[Brain::WdL] = WaL;
	rawInputCloseness[Brain::WdL] = WcL;
}
else
{
	rawInput[Brain::WdL] = 0.0;		//No activation at all.
	rawInputAngle[Brain::WdL] = 0.0;
	rawInputCloseness[Brain::WdL] = 0.0;
}

if(closestWarmRight != NULL)
{
	rawInput[Brain::WdR] = activation(oldFacing, newPosition, newFacing,
		closestWarmRight->getPosition(), WdR, animat->getViewRadius() +
		closestWarmRight->getDetectionRadius());
	rawInputAngle[Brain::WdR] = WaR;
	rawInputCloseness[Brain::WdR] = WcR;
}
else
{
	rawInput[Brain::WdR] = 0.0;		//No activation at all.
	rawInputAngle[Brain::WdR] = 0.0;
	rawInputCloseness[Brain::WdR] = 0.0;
}
```

```
if(closestFrozenLeft != NULL)
{
	rawInput[Brain::CdL] = activation(oldFacing, newPosition, newFacing,
		closestFrozenLeft->getPosition(), CdL, animat->getViewRadius() +
		closestFrozenLeft->getDetectionRadius());
	rawInputAngle[Brain::CdL] = CaL;
	rawInputCloseness[Brain::CdL] = CcL;
}
else
{
	rawInput[Brain::CdL] = 0.0;	//No activation at all.
	rawInputAngle[Brain::CdL] = 0.0;
	rawInputCloseness[Brain::CdL] = 0.0;
}

if(closestFrozenRight != NULL)
{
	rawInput[Brain::CdR] = activation(oldFacing, newPosition, newFacing,
		closestFrozenRight->getPosition(), CdR, animat->getViewRadius() +
		closestFrozenRight->getDetectionRadius());
	rawInputAngle[Brain::CdR] = CaR;
	rawInputCloseness[Brain::CdR] = CcR;
}
else
{
	rawInput[Brain::CdR] = 0.0;	//No activation at all.
	rawInputAngle[Brain::CdR] = 0.0;
	rawInputCloseness[Brain::CdR] = 0.0;
}

if(closestFemaleGiftLeft != NULL)
{
	rawInput[Brain::AdL] = activation(oldFacing, newPosition, newFacing,
		closestFemaleGiftLeft->getPosition(), AdL, animat->getViewRadius() +
		closestFemaleGiftLeft->getTemperatureRadius());
	rawInputAngle[Brain::AdL] = AaL;
	rawInputCloseness[Brain::AdL] = AcL;
}
else
{
	rawInput[Brain::AdL] = 0.0;	//No activation at all.
	rawInputAngle[Brain::AdL] = 0.0;
	rawInputCloseness[Brain::AdL] = 0.0;
}

if(closestFemaleGiftRight != NULL)
{
	rawInput[Brain::AdR] = activation(oldFacing, newPosition, newFacing,
		closestFemaleGiftRight->getPosition(), AdR, animat-getViewRadius() +
		closestFemaleGiftRight->getTemperatureRadius());
	rawInputAngle[Brain::AdR] = AaR;
	rawInputCloseness[Brain::AdR] = AcR;
}
else
{
	rawInput[Brain::AdR] = 0.0;	//No activation at all.
	rawInputAngle[Brain::AdR] = 0.0;
	rawInputCloseness[Brain::AdR] = 0.0;
}
```

```
if(closestFemaleNoGiftLeft != NULL)
{
	rawInput[Brain::UdL] = activation(oldFacing, newPosition, newFacing,
		closestFemaleNoGiftLeft->getPosition(), UdL, animat->getViewRadius() +
		closestFemaleNoGiftLeft->getTemperatureRadius());
	rawInputAngle[Brain::UdL] = UaL;
	rawInputCloseness[Brain::UdL] = UcL;
}
else
{
	rawInput[Brain::UdL] = 0.0;	//No activation at all.
	rawInputAngle[Brain::UdL] = 0.0;
	rawInputCloseness[Brain::UdL] = 0.0;
}

if(closestFemaleNoGiftRight != NULL)
{
	rawInput[Brain::UdR] = activation(oldFacing, newPosition, newFacing,
		closestFemaleNoGiftRight->getPosition(), UdR, animat->getViewRadius() +
		closestFemaleNoGiftRight->getTemperatureRadius());
	rawInputAngle[Brain::UdR] = UaR;
	rawInputCloseness[Brain::UdR] = UcR;
}
else
{
	rawInput[Brain::UdR] = 0.0;	//No activation at all.
	rawInputAngle[Brain::UdR] = 0.0;
	rawInputCloseness[Brain::UdR] = 0.0;
}

if(closestGiftLeft != NULL)
{
	rawInput[Brain::GdL] = activation(oldFacing, newPosition, newFacing,
		closestGiftLeft->getPosition(), GdL, animat->getViewRadius() +
		closestGiftLeft->getDetectionRadius());
	rawInputAngle[Brain::GdL] = GaL;
	rawInputCloseness[Brain::GdL] = GcL;
}
else
{
	rawInput[Brain::GdL] = 0.0;	//No activation at all.
	rawInputAngle[Brain::GdL] = 0.0;
	rawInputCloseness[Brain::GdL] = 0.0;
}

if(closestGiftRight != NULL)
{
	rawInput[Brain::GdR] = activation(oldFacing, newPosition, newFacing,
		closestGiftRight->getPosition(), GdR, animat->getViewRadius() +
		closestGiftRight->getDetectionRadius());
	rawInputAngle[Brain::GdR] = GaR;
	rawInputCloseness[Brain::GdR] = GcR;
}
else
{
	rawInput[Brain::GdR] = 0.0;	//No activation at all.
	rawInputAngle[Brain::GdR] = 0.0;
	rawInputCloseness[Brain::GdR] = 0.0;
}

if(closestMaleLeft != NULL)
```

```
{
	rawInput[Brain::MdL] = activation(oldFacing, newPosition, newFacing,
		closestMaleLeft->getPosition(), MdL, animat->getViewRadius() +
		closestMaleLeft->getTemperatureRadius());
	rawInputAngle[Brain::MdL] = MaL;
	rawInputCloseness[Brain::MdL] = McL;
}
else
{
	rawInput[Brain::MdL] = 0.0;	//No activation at all.
	rawInputAngle[Brain::MdL] = 0.0;
	rawInputCloseness[Brain::MdL] = 0.0;
}

if(closestMaleRight != NULL)
{
	rawInput[Brain::MdR] = activation(oldFacing, newPosition, newFacing,
		closestMaleRight->getPosition(), MdR, animat->getViewRadius() +
		closestMaleRight->getTemperatureRadius());
	rawInputAngle[Brain::MdR] = MaR;
	rawInputCloseness[Brain::MdR] = McR;
}
else
{
	rawInput[Brain::MdR] = 0.0;	//No activation at all.
	rawInputAngle[Brain::MdR] = 0.0;
	rawInputCloseness[Brain::MdR] = 0.0;
}

// If animat picked up a food-gift, make it remember how it was for later rewarding //
if(pickedUpGift)
	animat->rememberGift(rawInputCloseness[Brain::GdL], rawInput[Brain::GdL],
		rawInputCloseness[Brain::GdR], rawInput[Brain::GdR]);

///////////////// Load new input/goals and implement learning ////////////////
animat->implementLearning();

////////////////////////////// Bookkeping ////////////////////////////////////
if(animat->getSex() == 0)	//Separate males from females.
{
	if(animat->getTrack())
	{
		malesAlive++;
		maleFood += animat->getFoodEaten();
		malePoison += animat->getPoisonEaten();
		laidEggs += animat->getLaidEggs();
		rejections += animat->getRejections();
	}
	totalMales++;
}
else
{
	if(animat->getTrack())
	{
		femalesAlive++;
		femaleFood += animat->getFoodEaten();
		femalePoison += animat->getPoisonEaten();
		gifts += animat->getGifts();
	}
	totalFemales++;
}
```

```
animatNode = animatNode->getNext();             //Next animat in the list.

                }
        }

        ////////////////////////////// Write average to files ///////////////////////////
        if(malesAlive > 0)
        {
                fprintf(maleFile, "%f, %d, %d, %f, %f, %f, %f\n",
                        time, totalMales, malesAlive, maleFood/malesAlive,
                        malePoison/malesAlive, laidEggs/malesAlive, rejections/malesAlive);
        }
        if(femalesAlive > 0)
        {
                fprintf(femaleFile, "%f, %d, %d, %f, %f, %f\n",
                        time, totalFemales, femalesAlive, femaleFood/femalesAlive,
                        femalePoison/femalesAlive, gifts/femalesAlive);
        }
}
```