Genetic Algorithms and Neural Networks: The Building Blocks of Artificial Life

by Jacob Schrum

A Thesis Submitted in Partial Fulfillment of the
Requirements for Graduation with Honors in Computer Science

Southwestern University, Spring 2006
Approved

____
Walter Potter, Honors Advisor, Computer Science

____
Richard Denman, Committee Member, Computer Science

____
Max Taub, Committee Member, Biology

Contents

Introduction

What is it that makes something alive? The only types of organisms that we are aware of on earth are carbon based life forms. Despite the incredible diversity of life on this planet, all organisms share the same basic building blocks. We are made of microscopic cells composed primarily of carbon molecules. Some organisms consist of a single cell, while others are collections not only of many cells but many different types of cells. The only known entities that in some ways challenge this notion of cellular life are viruses. Viruses are generally not considered living because they can only reproduce themselves with the aid of host cells, yet in spite of this they share many qualities with what are considered "living" organisms: they are carbon based, they reproduce and they exhibit inheritance. Because viruses exhibit inheritance, they are capable of evolving.

The process of evolution is very important to the history of life on earth. Biologists theorize that all organisms on earth descended from either a single common ancestor or gene pool. If true, this would mean that the diversity of life on earth was purely the result of evolution. We know that evolution occurs; we can observe it in a laboratory or in nature. What we cannot do is observe the path of evolution over the billions of years that life is believed to have existed on this planet. Archeological evidence provides some clues as to the course of evolution, but such evidence is obviously lacking in many ways. Fossils do not move. They are no longer alive and can only tell us so much.

One way to fill in the gaps is through the use of complex computer models that attempt to reconstruct and simulate the past. Plausible models of animal behavior can be made based on fossil evidence. However, by abstracting away from certain details and focusing on others, more generalized models that attack the fundamental questions of life can be created. What is life? What are the most basic elements required to consider a system alive? How can we measure evolution? The field of scientific enquiry that strives to answer these questions with the use of such models is the field of Artificial Life.

According to Christopher Langton, one of the pioneers of the field: [6, pg. 1]

> Artificial Life is the study of man-made systems that exhibit behaviors characteristic of natural living systems. It complements the traditional biological sciences concerned with the *analysis* of living organisms by attempting to *synthesize* life-like behaviors within computers and other artificial media. By extending the empirical foundation upon which biology is based *beyond* the carbon-chain life that has evolved on Earth, Artificial Life can contribute to theoretical biology by locating *life-as-we-know-it* within the larger picture of *life-as-it-could-be.*

This work describes an Artificial Life simulation that addresses the questions of how evolution gives rise to diversity among organisms, and how speciation can occur in a closed environment. The details of the simulation's construction as well as its outcomes are described in this work.

Chapter 1

Design

1.1 Purpose

The primary purpose of this work is to create a simulation in which diversity can arise in an evolving population of animal-like organisms. For the purposes of this work, diversity will be defined as the persistent presence in a population of at least two types of animals (preferably more) that are adapted to survive in observably different manners. A persistent presence could mean that two species evolve and then remain in equilibrium with one another for the duration of the simulation, or that new species continually evolve and replace common ancestor species for as long as the simulation runs.

Of particular interest is whether or not an initially homogeneous population can give rise to such diversity within an enclosed world. The capacity for evolution will be built into the system, but diversity will only arise if populations of different species emerge and then diverge. It is possible for the entire population to evolve as a whole and therefore remain homogeneous even though it is evolving. Such a situation would not give rise to diversity.

The process by which one species evolves into two or more species is called speciation. The most thoroughly documented form of speciation is allopatric speciation, in which a single population of organisms that has somehow become geographically separated into two populations then evolves into two different species. Should the two populations be reunited they would then be unable to interbreed. Another type of speciation is parapatric speciation, in which there is no geographical separation between regions, but none the less a population separates into multiple populations due to the large size of the region. Some interaction between populations still occurs, but in general they remain separated to the extent that they can eventually no longer interbreed. Yet another form of speciation is sympatric speciation, which is slightly more controversial than the other two types because fewer examples of it are apparent. Sympatric speciation is the evolution of separate species from a single population within a relatively small geographic area. This is controversial because it is assumed that any minor differences in individuals would quickly disappear or become common throughout the population as a result of interbreeding before the population would have the chance to speciate. Because of this, when sympatric speciation does occur it is assumed to be the result of the evolution of specialized mating preferences (for sexually reproducing organisms).

Sympatric and parapatric speciation are the two types of speciation that are of interest to this study. Animals will be able to freely move between all regions of the environment, which does not allow for allopatric speciation to occur. It is possible that the environment could be large enough for parapatric speciation to occur, but not certain. If the environment is not large enough, then only sympatric speciation would be capable of occurring. The emergence of diversity within the population will therefore depend on sympatric and/or parapatric speciation.

The final purpose of this work is to demonstrate the usefulness of both genetic algorithms and neural networks in artificial life simulations. These tools, which are themselves inspired by the organisms of our earth, have long been used in artificial life simulations to model new organisms within the environment of the computer. This simulation demonstrates a novel way to combine the two in a single multi-agent simulation.

1.2 World

1.2.1 Environment

The environment in which the simulation plays out is a very simple one indeed. The landscape is completely bland and devoid of obstacles and distinguishing characteristics. The only interesting elements of the environment are the organisms that inhabit it, namely plants and animals.

Although the world is completely enclosed it does not have any boundaries. Positions in the world are referenced as points on a Cartesian coordinate plane, but the edges of the plane wrap around to connect with each other, so that the environment is effectively the surface of a torus. All inhabitants of the world are completely oblivious to this fact. Plants do not grow any differently near the edges of the plane than anywhere else and animals take no notice when they pass the boundary from one edge of the plane to the other.

The only unusual feature of the environment was inserted initially as a means to reduce computation and free up computer resources, but this aspect also has an interesting effect on the way animals sense their world. The Cartesian plane that defines the environment is partitioned into several rectangular units referred to as `Bin`'s. An animal cannot sense the `Bin`'s in the environment, but its senses are affected by their presence. An animal cannot sense anything outside of the `Bin` it currently occupies. This gives animals a blind spot near the edges of `Bin`'s. The reasons for this configuration are explained in detail within the *Bins* module of the *Code* section of this document.

1.2.2 Plants

In the world there are many types of organisms, and among these are plants. Plants are often at the bottom of any given food chain, and this simulation is no different (though one could just as easily model an ecosystem independent of plant life). Rather than have multiple types of plant species, there is only a single type of plant that can be eaten by all herbivores.

In the context of this simulation, plants are nothing more than a food source. Although real world plants do evolve in a number of interesting ways, the plants of this simulation do nothing but grow and be eaten. The primary purpose of this simulation is to study the evolution of animal-like organisms. Therefore the plant aspect of the simulation is simplified to remove complications. Reducing the complexity of the plant model frees up computer resources for the more complex calculations needed to manage the world's animal population.

As stated above, plants are nothing but a food source. All plants are identical in that each takes up the same amount of space in the world and provides the same amount of energy to animals upon being eaten. Plants do not actually grow so much as instantly appear in the world. Animals can sense plants, and some animals can eat plants, so the presence of plants can definitely influence an animal's behavior. However, plants serve no additional role in the environment. They do not directly affect the movement speeds of animals, nor can they serve as cover for an animal of prey attempting to escape a predator. They are quite limited in all respects. Details of how plants are implemented in the world are further discussed within the *Plants* module of the *Code* section of this document.

1.2.3 Animals

The animals are the primary focus of this simulation, and as such have been given the most attention. Although artificial lifeforms with very simple designs can be interesting in their own right, the animals of this simulation are quite complex. Their design is the most sophisticated aspect of this simulation.

Every animal in the simulation has its own personal neural network, which controls the animal's movements in the environment. The prominent features of the environment that an animal senses (all of which are other organisms) are encoded into a form that the animal's neural network can understand. These inputs are passed through the neural network and result in outputs which control the animal's behavior. The particular type of neural networks used in this simulation simulate both long term memory and learning, through the use of Hebbian learning, and short term memory, through the use of discrete time delays on neural network synapses. More information on what neural networks are, how they work, and how they are implemented in this simulation is provided in the section titled *Neural Networks Supporting Discrete Time Delays and Hebbian Learning*.

The composition of an animal's neural network as well as several other pertinent traits related to its behavior are stored within an animal's chromosomes. These chromosomes are used in conjunction with a genetic algorithm whenever two organisms mate to produce offspring. The genetic algorithm performs the operations of crossover and, with a certain probability, mutation on chromosomes during mating in order to produce new chromosomes for offspring. The crossover operation gives rise to new combinations of preexisting genes already within the population, and mutation, when it occurs, creates completely new genes, which could be potentially detrimental or advantageous to those that possess them. The genetic algorithm used in this simulation is described in more detail in the *Genetic Algorithms* section within the chapter named *Code*.

At the beginning of the simulation a starting population of organisms is randomly scattered throughout the world. This population is homogeneous in every trait except for sex, since members of both sexes are needed in order for sexual reproduction to occur. Because it seems to be the most basic of dietary behaviors, the initial population is composed entirely of herbivores. It is however possible for predation, cannibalism, and even carrion eating to evolve throughout the course of the simulation. Animals and their various traits and behaviors are described in full detail within the *Animals* module of the *Code* chapter.

1.3 Haskell

1.3.1 Language Choice

The language in which the simulation is programmed in is Haskell, a pure functional programming language. As a functional programming language, Haskell has a larger overhead in terms of operating costs compared to imperative languages such as C and C++. That aside, many of Haskell's features make it ideal for this simulation. The manner in which Haskell evaluates commands, the many unique features it offers, and the general programming methodology of functional programming languages such as Haskell make it very well suited to the needs of the simulation.

Haskell features lazy evaluation, which means that expressions are not evaluated until absolutely necessary. An argument to a function that is itself an expression will remain an expression until further execution becomes impossible without evaluating the expression. The advantage of this is that some expressions never need to be evaluated, thus preventing the computer from wasting time on calculations that are not needed. Once an expression is evaluated, its result is stored and remains available until it is no longer needed. This means that the expression never needs to be evaluated more than once.

Yet another advantage of lazy evaluation is that it allows for recursively infinite data types. Streams (infinite lists) are the most common infinite data type, and they are used extensively throughout the simulation. Data types can be infinite because Haskell only evaluates a data structure's contents as needed. The remainder of the structure is not evaluated until needed, at which point the recursive definition of the structure is used to obtain that next portion of it.

Other particularly useful features of the Haskell programming language are anonymous functions and list comprehensions. Anonymous functions are simple functions meant to be used only once within a single function. They are generally very simplistic, yet at the same time very specific, and therefore often only need to be used once. Because of this there is no need for them to have names. Because they have no names, anonymous functions cannot be recursive. In this simulation they are often used to recast data from one form to another. List comprehensions are useful list constructing tools that utilize syntax similar to that used in set theory. A list comprehension is defined as a collection of all elements taken from another list such that a series of predicates are satisfied. This powerful tool makes it easy to both construct lists and filter out elements from other lists. The syntactical similarity between list comprehensions and set theory makes list comprehensions easy to understand.

Haskell is also well suited to this simulation simply by virtue of being a functional programming language. Oddly enough, functional programming languages are often defined in terms of what they cannot do: a pure functional language does not have any assignment statements, nor does it have any side-effects [3]. When a variable takes on a value it maintains that value throughout execution. This means that the variables are referentially transparent. Put another way, this means that any variable name can be replaced by its value without changing the result of execution, because the variable cannot be changed once set. This eliminates many errors that arise in imperative languages as a result of uncertainty about the current state of the machine and the variables it maintains. Functional languages also lack flow control. A functional program is a function, which itself calls other functions until at the lowest level nothing but language primitives are applied. Execution consists of a series of evaluations rather than a list of instructions to be performed in sequence, as occurs in imperative languages.

There are some situations in which the state of the machine is of vital importance to the execution of a program, such as when reading input files. In these situations Haskell makes use of Input-Output (IO) data, which is restricted in how it can be used. These restrictions maintain the functional purity of Haskell (other functional languages such as LISP mix functional and non-functional features indiscriminately).

The nature of functional languages like Haskell is also such that code is more modular and reusable. At the same time, more can generally be accomplished with less code. The great deal of code created to complete this project may make one doubtful of such a claim, but rest assured that Haskell's lack of assignment statements and its modularity make the code much more compact and easier to understand than if it had been programmed in an imperative language. For all of these reasons, Haskell is the language of choice for this simulation.

1.3.2 Compiler Choice

There is not much in the way of selection when it comes to compilers for Haskell. Haskell is often used to write very small programs that can be easily run using an interpreter. The Haskell interpreter *Hugs* is perhaps the most popular. *Hugs* was used at many stages of development for testing of individual modules, but because it is an interpreter it cannot provide the speed necessary to run such a massive simulation.

The three common Haskell compilers available on the internet are *GHC* (Glasgow Haskell Compiler), *nhc98* and *HBC* (Chalmers Haskell-B Compiler). Support for *HBC* is fairly low, and it has not been updated in years, which ruled it out as a viable candidate. Both *GHC* and *nhc98* are fairly popular, but *GHC* is capable of producing faster code, at the expense of longer compile times, which is a small price to pay. Therefore *GHC* was used to compile the code for the simulation.

Chapter 2

Code

2.1 A Note on Style

In the world of programming, more and more importance is being given to documentation of code. It is not enough to simply have code that works. One must also explain how it works and why certain methods are used in favor of others. An important aspect of documenting one's code is commenting it. Haskell supports comments just as any other programming language, though it also has built in support for literate programming. Literate programming is a term coined by Donald Knuth [5]. It is a style of programming in which the program essentially becomes an essay that describes to a reader of the code what the programmer is asking the computer to do.

Haskell's literate programming mode assumes from the outset that a programmer will thoroughly document the code, and therefore allows such documentation to appear in the source files free of any special comment characters or delimiters. In contrast to this, all lines of code must start with a greater-than symbol (>) to be recognized as code by the compiler. The result is a side-by-side combination of code and documentation that helps a reader to understand both what the computer is being asked to do and why the programmer asked it to do that.

All Haskell modules for this project were written in the literate style. The original Haskell modules have become sections of the *Code* chapter of this thesis. Whenever a module name is referenced in a Haskell `import` statement, a comment provides the section name in this document that corresponds to that module name.

The literate text creates a significant portion of this thesis. Topics are introduced and elaborated upon within the modules/sections, such that all code pertaining to a certain topic is presented to the reader along with information regarding the topic. For the most part the volume of text is greater than that of the code in any given module, though there are a few exceptional cases when the amount of code is vast, and therefore the code itself is quite complex.

Furthermore, the code has been slightly modified for the sake of presentation. If copied directly the code would not work, because many of the special symbols of Haskell have been converted into mathematical symbols whose meanings are universally understood. Also, subscripts have been used for the names of some of the variables in the code, which is not possible in Haskell. For those that may wish to convert the following code into syntactically valid Haskell source files, the following table of Haskell symbols and their equivalents within this document can be used (the code is also on the included CD).

The following modules were built from the ground up, and are therefore arranged in this order for the sake of presentation. The first few modules deal with very general concepts such as random numbers, probability and abstract data types. Later modules begin to focus on the simulation itself. The results of analyzing the simulation are reported in the *Results* chapter.

Haskell Symbol	Mathematical Equivalent
$->$	$\rightarrow$
=>	$\Rightarrow$
>=	$\geq$
<=	$\leq$
==	$\equiv$
&&	$\wedge$
||	$\vee$
*	$\times$
x^ n	x^n
\	λ
‘div‘	`div`
‘mod‘	`mod`
[0..]	$[0, \ldots]$

Haskell symbols and their pure mathematical equivalents

2.2 Randomness

2.2.1 Overview

The two modules in this section deal with randomness, which is in some ways paradoxical in the context of a computer program. Computers cannot generate randomness, but the following two modules can simulate it. The section titled *Streams of Random Numbers* provides a means of creating infinite lists (streams) of pseudo-random numbers. The section titled *Probability* provides a way to define probability distributions, which are used to pick which events occur given a list of disjoint possibilities.

2.2.2 Streams of Random Numbers

Random numbers can be quite a controversial subject. One can argue that it is possible to predict everything given enough knowledge. That would mean that randomness is simply an abstraction for things we cannot understand, or for which there is not enough time or processing power to make carrying out the necessary calculations worthwhile, even if possible.

In computers the issue of randomness is somewhat less philosophical but still important. Strictly speaking, computers are completely deterministic. This means that generating random numbers is not possible. However, it is possible to generate pseudo-random numbers, which share many of the properties of random numbers and for nearly all applications are equally useful.

In most imperative languages programmers have the option of creating their own random number generators with seeds, which are initial numbers that influence the series of random numbers created by the generator. Some languages simplify this further and use the value from the computer's internal clock as a seed. The problem with this practice is that one cannot replicate an experiment if the clock is used for seeding.

In Haskell the deterministic nature of computers becomes even clearer. Haskell also has a built in random number generator, but because it is dependent upon the state of the computer it returns an IO type. This makes computations requiring a large number of random numbers difficult without the use of `unsafePerformIO`, which converts an IO type to a pure type. In the interest of maintaining the functional integrity of code, this function is not used. Instead, a custom random number generator taken from the book "The Craft of Functional Programming" by Simon Thompson [9] is used.

The method presented in Thompson's book is called the "linear congruential method," and all of the random number streams presented in this module stem from this method.

```
>module RandomStream
> (nextRandomIntegral,
> randomIntegrals, boundRandomIntegrals,
> smallRandomFloatings, boundRandomFloatings) where
```

The three constants below come directly from Thompson's book, and each plays a role in the generation of pseudo-random numbers. The `nextRandomIntegral` function below takes an `Integral` value as input and multiplies it by the constant named `multiplier`. This value is then added to the `incrementer`, and modulus division by the constant `modulus` is performed on the result. The value `modulus` is important because it sets the range for the function. All numbers generated by this function have a range [0, `modulus`). The `modulus` is set to 65536, which is more than large enough for all applications to which this random number generator is applied.

```
>multiplier ::  Num a ⇒ a
>multiplier = 25173

>incrementer ::  Num a ⇒ a
>incrementer = 13849

>modulus ::  Num a ⇒ a
>modulus = 65536
```

According to Donald E. Knuth's book "The Art of Computer Programming Vol. 2: Seminumerical Algorithms," [4] Thompson's choice of "magic numbers" is valid. Knuth's book (published in 1969) deals with, among other things, the issue of pseudo-random number generation by computers. Knuth explains that the linear congruential method always creates a looping sequence of numbers, and while it would be desirable to have an infinite list of random numbers, one must instead settle for creating a sequence with as large a period as possible. The period is the number of values generated before looping occurs.

The maximum possible value of the period is `modulus`, in which case every value in the sequence occurs exactly once before repeating. In order for the period to equal `modulus`, several conditions must be met, as according to the Theorem A in Knuth's book:

1. `incrementer` must be relatively prime to `modulus`.
2. (`multiplier` - 1) is a multiple of p, for every prime p dividing `modulus`.
3. (`multiplier` - 1) is a multiple of 4, if `modulus` is a multiple of 4.

Thompson's numbers satisfy condition 1 because $\gcd(13849, 65536) = 1$. Condition 3 is satisfied because $4 \mid (25173 - 1)$ and $4 \mid 65536$. Proving condition 2 is more difficult, but thankfully Knuth provides a simpler test that can be used to assure that `multiplier` and `modulus` are chosen so that every value in the range [0, `modulus`) occurs before repeating. Since 65536 is a power of 2 ($65536 = 2^{16}$), all that must be done to assure that every number in the sequence occurs before repeating is to choose `multiplier` such that `multiplier mod` $8 = 5$. Thompson's `incrementer` satisfies this property (25173 `mod` $8 = 5$).

One advantage of having the period equal `modulus` is that it does not matter what value is used to seed the generator. Since all values from 0 to (`modulus` - 1) appear in the sequence exactly once, any of these values can be used as a starting point. This is convenient because multiple copies of this generator are meant to be used simultaneously, so each needs to have a different starting seed in order to be independent from the others.

Now on to the actual random number generator. Notice that the type signature of `nextRandomIntegral` allows for the generation of any general `Integral` type. This means the function works for both `Int` and `Integer` types, which is convenient since random numbers of both types are needed by the simulation.

Inputs:

- seed or previous random number

```
>nextRandomIntegral ::  Integral a ⇒ a → a
>nextRandomIntegral n = (n × multiplier + incrementer) mod modulus
```

The `nextRandomIntegral` function takes a single value as input and generates another. In order to obtain many random numbers this function needs to be applied repeatedly, and Haskell happens to have a useful function named `iterate` which accomplishes this. Iterating with the `nextRandomIntegral` function means that the result of each application of `nextRandomIntegral` is itself used as the input to the next application of the function. The outputs are returned in a stream. This is possible because of Haskell's use of lazy evaluation. All that is needed is an initial value to start with, which is the seed of the random number generator. The seed is the input to the function `randomIntegrals` and different seeds create different random streams. According to Thompson the numbers in this sequence "all occur with the same frequency." [9, pg. 367]

Inputs:

- seed

```
>randomIntegrals ::  Integral a ⇒ a → [a]
>randomIntegrals = iterate nextRandomIntegral
```

Thompson also provides a method of restricting the range of this function to generate numbers in the range [s,t]. This is done with the `scaleSequence` function. It works by splitting the original range of numbers into "range blocks" and then mapping values in given ranges to new values. Unfortunately, it only works properly if the size of the new range is divisible by `modulus`. Otherwise it maps the last few values in the initial range [0, `modulus` - 1] to values outside of the desired range [s,t]. Therefore the function `boundRandomIntegrals` removes values out of range after applying `scaleSequence` to the initial stream of random numbers produced by `randomIntegrals`.

Inputs:

- minimum value
- maximum value
- list of random integrals in range [0, `modulus` - 1]

```
>scaleSequence ::  Integral a ⇒ a → a → [a] → [a]
>scaleSequence s t = map scale
>    where
>       scale n = n div denom + s
>       range = t - s + 1
>       denom = modulus div range
```

The `boundRandomIntegrals` function combines the `randomIntegrals` and `scaleSequence` functions into one simple function, and then removes the occasional value that is outside of the scaled range. All values within the desired range are equally likely to occur even though use of the `scaleSequence` function may result in some values being discarded. The probability of `scaleSequence` returning a value that needs to be discarded is slightly less than the probability of any particular valid value being returned. After invalid values are discarded the resulting distribution of possible random numbers is uniform. Calling this function with a desired range [a,b] generates a stream of numbers in this range. The seed for the random number generator is also needed.

Inputs:

- minimum value
- maximum value
- seed

```
>boundRandomIntegrals :: Integral a ⇒ a → a → a → [a]
>boundRandomIntegrals a b n = filter (≤ b) (scaleSequence a b (randomIntegrals n))
```

Random floating point numbers are also needed in the simulation. Thompson does not provide a method for generating random floating point numbers, but it is quite easy to derive a floating point number generator from the integral number generator he provides. First a function named `smallRandomFloatings` that generates floating point numbers in the range [0,1] is defined. This generator is particularly useful on its own, and also serves as the basis for the function `boundRandomFloatings` further below. The `smallRandomFloatings` function works by dividing every number in the `randomIntegrals` sequence by the maximum possible value in that sequence, namely `modulus - 1`. This allows for `modulus` number of possible random numbers in the range [0,1], which is more than enough for the purposes of the simulation.

Inputs:

- seed

```
>smallRandomFloatings :: Floating a ⇒ Integer → [a]
>smallRandomFloatings n = map ((/ (modulus - 1)).fromInteger) (randomIntegrals n)
```

Since random floating point numbers larger than 1 and less than 0 are also needed in the simulation we need a function that scales floating point numbers to this range. Unlike the `boundRandomIntegrals` function which scales the range of `randomIntegrals` down, the `boundRandomFloatings` function below scales the range of `smallRandomFloatings` up. It can map values from this function to any range, but it is important to remember that no matter what the range, there are still only `modulus` number of possible values. This means that the number generator below works best for small ranges in which high precision is not important. As such it is ideal for this simulation. However, this generator would not work well for generating floating point numbers in particularly large ranges, as increasing the range increases the space between consecutive values that this function is capable of generating. In fact, if the range is set to [0, `modulus - 1`] then the space between each possible value is exactly one. In other words, the usefulness of the function `boundRandomFloatings` decreases as the range increases. The function works by multiplying each value of the `smallRandomFloatings` stream by the range, thus creating a stream with values from 0 to $a - b$, where a is the minimum and b is the maximum. The values of this stream are then shifted by being added to the minimum value. This makes the smallest possible value equal to the minimum of the range, even if it was negative, and the greatest possible value equal to the maximum of the range.

Inputs:

- minimum value
- maximum value
- seed

```
>boundRandomFloatings :: Floating a ⇒ a → a → Integer → [a]
>boundRandomFloatings lo hi n = map ((+ lo).(× (hi - lo))) (smallRandomFloatings n)
```

2.2.3 Probability

The probability of an event is the likelihood that it will occur. Probabilities are defined to be within the range [0,1], with 0 meaning the event will never occur and 1 meaning the event will always occur. Probability comes into play when there is some degree of uncertainty.

Probability is the mathematical modeling technique that allows us to deal with this uncertainty. One can argue that given enough data and computing capacity one could accurately determine the outcome of any given situation, but because such data and computing capacity are often not available, we use probability to predict the likelihood of possible outcomes. However, a computer is a completely deterministic environment, and we have complete control over all possible outcomes. The uncertainty that gave rise to the field of probability does not apply to computation on a computer, but because this simulation is partially modeled on the real world, we want a degree of uncertainty. We want some events to be more likely to occur than others, but nothing should be completely certain. The purpose of this module is to model pseudo-uncertainty, and to create a mechanism for determining which of several potential events occurs, given the context of this pseudo-uncertainty.

The pseudo-uncertainty is provided by the use of pseudo-random numbers. The event determining mechanism is defined by the functions below. They assume a situation in which exactly one event is chosen from among at least two events that occupy the same discrete probability space. All events are disjoint with probabilities greater than 0. Furthermore, the sum of the probabilities of all events in the space is 1. A probability distribution representation is made that associates each event in the probability space with an interval inside the range [0,1], and this representation is used in conjunction with uniform random numbers in the same range to choose which of several possible events occurs.

```
>module Probability
> (ProbEvent,
> makeDistribution,
> pickEvent) where
```

The `ProbEvent` data type represents a single event and its assigned interval in a probability distribution representation. All instances of `ProbEvent` are of type `E`, which contains the members `event`, `startProb` and `endProb`. The event is any valid type, and the start and end probabilities are `Float` values which must be in the range [0,1]. They are the lower and upper bounds of the interval associated with the given event. The following invariant must also hold: `endProb` > `startProb`. The given interval is a half-open interval that is closed at `endProb` and open at `startProb`. The only exception to this is the first interval in any given distribution, which is closed at 0 in addition to being closed at its `endProb` (whatever that may be). This is acceptable because the probability of a single point is zero, and can therefore be added to a given interval without upsetting the distribution. This is necessary so that the union of the intervals is equal to [0,1] instead of (0,1]. The size of any given interval is the probability of the associated event.

```
>data ProbEvent a = E {event::a, startProb::Float, endProb::Float}
>    deriving (Show, Eq)
```

A single `ProbEvent` has little use, but a list of them, properly arranged and having certain properties, represents a probability distribution. A list of properly formed `ProbEvent` types essentially represents a table of associations: for each `ProbEvent` the values (`startProb`,`endProb`] are associated with the `event` data member. For this to work, one `ProbEvent` instance in the list must have a `startProb` of 0 and another must have an `endProb` of 1. Also, each `endProb` value of all the other `ProbEvent` instances must be equal to the `startProb` value of another instance, and none of the intervals can overlap. For the sake of simplicity, the list must also be ordered in increasing order of start and end probabilities. This final property is not necessary for representing a probability distribution, but maintaining it makes the upcoming `pickEvent` function easier to define.

Such a list is easily derived from a list of events and a list of corresponding probabilities, but certain problems do not start out formulated in these terms. Sometimes the likelihood of different events occurring is defined in terms of arbitrary quantities whose values are proportional to the probabilities of the given events. Given a list of such quantities for all events of a given probability space, one can determine the probability of each event by dividing its associated quantity by the sum of all quantities in the list. Given these probabilities, a representation of a probability distribution for the probability space can be constructed.

First we will demonstrate the mathematical correspondence between our given probability distribution representation and a random variable equivalent to it. This random variable is a function $\mathcal{Y} : \Re \rightarrow \Re$ that produces the output we are interested in. We assume it to have a probability density function f and a cumulative distribution function F. This implies

$$P(u \leq \mathcal{Y} \leq v) = F(v) - F(u) \text{ where } F(w) = \int_{-\infty}^{w} f(t)dt$$

Because F is a cumulative distribution function, it is injective, surjective and continuous. Therefore for $F : \Re \rightarrow [0,1]$ there is an inverse $G : [0,1] \rightarrow \Re$. In other words, $G(w) = F^{-1}(w)$. Now we take another random variable $\mathcal{X} : \Re \rightarrow [0,1]$. This variable is our random number generator, and is therefore uniformly distributed.

$$\begin{aligned} & P(a \leq G(\mathcal{X}) \leq b) \\ = \; & P(a \leq F^{-1}(\mathcal{X}) \leq b) \\ = \; & P(F(a) \leq \mathcal{X} \leq F(b)) \\ = \; & F(b) - F(a) \qquad \{\mathcal{X} \text{ is uniformly distributed}\} \\ = \; & P(a \leq \mathcal{Y} \leq b) \end{aligned}$$

Therefore $\mathcal{Y} = G(\mathcal{X})$ in distribution, where $\mathcal{X}$ is a random number generator that produces values in the range $[0,1]$. The function G is our probability distribution representation. It takes a randomly generated number in the range $[0,1]$ and returns the corresponding event. The function that picks this event out of the probability distribution representation is `pickEvent`, but first we define the function that creates these representations, namely `makeDistribution`.

The `makeDistribution` function takes a list of events and a corresponding list of arbitrary quantities as input. These quantities are converted to probabilities and then assigned a range within [0,1]. For each quantity q, this range is (`startProb`, `endProb`] for the event, such that `endProb - startProb` = q/S, where S is the sum of all quantities in the list. In other words, the difference between `startProb` and `endProb` is the probability of the event occurring.

Inputs:

- list of events
- list of values on a common scale

```
>makeDistribution ::  [a] → [Float] → [ProbEvent a]
>makeDistribution as bs = buildDistribution as bs 0
> where
>    s = sum bs

>    buildDistribution ::  [a] → [Float] → Float → [ProbEvent a]
>    buildDistribution [] [] _ = []
>    buildDistribution (a:as) (b:bs) lo
>       = (E {event = a, startProb = lo, endProb = hi}):(buildDistribution as bs hi)
>          where
>             hi = lo + (b / s)
```

Once such a distribution is made, one needs a mechanism to select random events. Using a probability distribution representation, the function `pickEvent` takes a number in the range [0,1] and returns the associated event. We define a *determining factor* to be such a number, namely a number in the range [0,1] that determines which event is picked from a probability distribution. The probability distribution is a list of ProbEvent types arranged in proper order. The number from the range [0,1] that the function takes as input is presumably a random number from a uniform distribution, namely the random number generator, but the function itself does not assure that this is the case. It works by taking advantage of the fact that the list representing the probability distribution is ordered by increasing probability ranges, and goes through the list until it finds the range within which the random number given as input lies.

Inputs:

- list representing a probability distribution
- a number (presumably random) in the range [0,1], known as a determining factor

```
>pickEvent ::  [ProbEvent a] → Float → a
>pickEvent (e:es) p
> | p ≤ endProb e = event e
> | otherwise = pickEvent es p
```

2.3 Abstract Data Types

2.3.1 Overview

There is a general dearth of abstract data types (ADT's) for functional languages such as Haskell. There are many common data structures used frequently in imperative languages that work differently in functional languages. The semantics of a functional languages are very different from those of an imperative language, therefore it is not good enough to simply translate imperative data structures into functional ones. However, this is what many people used to do.

Now there are several data structures available that are designed specifically for functional languages. Some are functional adaptations of common imperative data structures, and others are structures unique to functional languages that take advantage of their unique properties. One of the pioneers of this field is Chris Okasaki, whose book "Purely Functional Data Structures" [7] describes many useful data structures, some of which have been adapted for use in this project.

The first module of this section defines a functional queue data type (one of the most basic ADT's in computer science). The next module defines a Binary Random Access List (BRAList), which is a data structure unique to functional programming that represents data similarly to how binary numbers are represented. The resulting structure allows for both quick random access (a feature of imperative arrays that functional lists sadly lack) and quick list operations. These BRAList's are used to create Binary Random Access Matrices (BRAMatrix's), which are defined in the third module.

2.3.2 Queue

A queue is a First-In-Last-Out (FIFO) data structure. It is a collection of objects with a front and an end. Objects are added to the end of the structure and removed from the front of the structure. Queues are common data structures in computer science, that are quite easy to model with lists (head to dequeue; concatenate at the end of the list to enqueue). However, to model a queue in such a way within Haskell is grossly inefficient since all operations on the rear of the list require passing through the entire list, costing linear time. Therefore a more efficient functional method is needed.

Such a method is provided in Okasaki's book. Okasaki describes several different types of queues for various applications.

Of particular importance is whether or not the data structure need be persistent. Persistent data structures are those for which multiple versions of the same instance need to be available at the same time. This can be complicated with imperative data structures, which normally use destructive updates, but functional data structures are persistent by default. However, functional data structures can still be used in a strictly non-persistent manner. Either way it is important that the application match the structure so as to maximize performance.

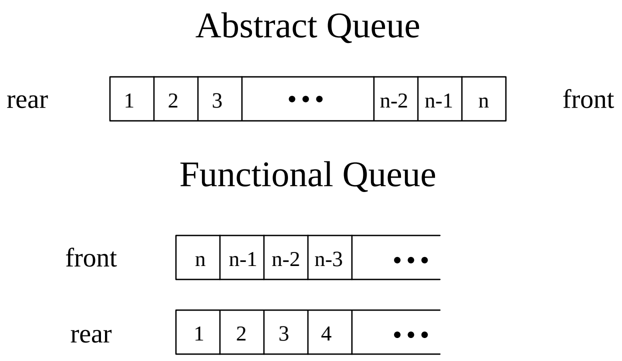
Another issue of performance is whether or not amortized bounds are acceptable. An amortized bound is normally smaller than a big-O bound, but it only applies over a series of operations. It is possible for individual operations to be either worse or better than the amortized bound, but the relative frequency of these different types of operations is such that the low cost operations make up for the high cost operations.

The queue presented below is intended to be used in a non-persistent manner, and amortized bounds are acceptable. It does not function efficiently when multiple versions of the same instance are required, but it "cannot be beat for applications that do not require persistence and for which amortized bounds are acceptable," [7, pg. 44] according to Okasaki.

Such a queue is ideal for the purposes of this project. These queues are used within time delayed neural networks in order to process input values for each synapse. The details of how and why this is done are explained in the section titled *Neural Networks Supporting Discrete Time Delays and Hebbian Learning*. It is possible for each neural network to have such a queue on every synapse, so it is important that they be fast. Because only one version of each queue exists at any given time, persistence is not necessary. Amortized bounds are acceptable because the overall speed of execution is more important than the speed of individual operations. The performance of the queue over a series of operations makes up for the occasional slow operation.

```
>module MyQueue
> (MQueue, queueFront, queueRear, queueSize,
> createQueue, isQueueEmpty,
> enqueue, peek, dequeue,
> listQueue, rebuildQueue) where
```

Abstractly, a queue is a list to which elements are added at one end and removed at the other. Such a queue is represented below by two lists: one named `queueFront`, which contains the elements near the front of the queue, and `queueRear`, which contains the elements near the rear of the queue stored in reverse order. If `queueFront` was concatenated to the reverse of `queueRear`, then the result would be the abstract queue mentioned above. The queue's size is also stored for the sake of convenience (the value of `queueSize` could easily be calculated at any time by adding the lengths of the two lists).



Relation between the abstract idea of a queue and its representation within this module. The positions of elements within the queues are labeled from 1 to n, which could be any integer value. Remember that although the queue is implemented as two lists, these lists do not need to be of the same size.

The justification for such a representation is that most dequeues and enqueues can be performed in constant time. An enqueue is accomplished by putting the new element at the head of the rear list, and a dequeue is accomplished by taking the head of the front list. The only problem arises when the front list is empty, in which case the rear list is reversed and is used to replace the front list, leaving the rear list empty. This is an expensive operation, but its cost and frequency still allow for a constant amortized time (see Okasaki's book for the proof [7, pg. 43-44]). Therefore, these queues maintain the invariant that the front list can never stay empty unless the rear list is also empty.

```
>data MQueue a = Q {queueFront::[a], queueRear::[a], queueSize::Int}
>    deriving (Show, Eq)
```

The `createQueue` function takes a list (an abstract queue) and makes a proper `MQueue` from it by making it the value of `queueFront`.

Inputs:

- abstract queue list of any type

```
>createQueue :: [a] → MQueue a
>createQueue f = Q {queueFront = f, queueRear = [], queueSize = length f}
```

The `isQueueEmpty` function returns `True` if the queue is empty and `False` otherwise. It does this by checking the `queueFront` list only, because as long as the invariant is maintained there will never be elements in the rear list when the front list is empty.

Inputs:

- an `MQueue` instance of any type

```
>isQueueEmpty :: MQueue a → Bool
>isQueueEmpty q = null (queueFront q)
```

To enqueue an element means to add it to the end of the queue. In the `MQueue` representation this simply means adding the element to the front of the `queueRear` list and increasing the queue's size by one. The exception to this is when the queue is empty, meaning that adding an element to the rear list would break the invariant. Therefore, the new element is instead added to the front list in the case that the queue is empty. This means that `enqueue` maintains the invariant and performs in constant amortized time in both cases.

Inputs:

- an `MQueue` instance of any type
- an element of the same type

```
>enqueue :: MQueue a → a → MQueue a
>enqueue (Q [] [] 0) x = Q [x] [] 1
>enqueue q x = q {queueRear = x:(queueRear q), queueSize = queueSize q + 1}
```

Peeking at the front of a queue returns the front element without actually changing the queue itself. This means returning the head of the front list of the queue. If the queue is empty, then attempting to peek at the front will cause an error, because the head of an empty list cannot be taken.

Inputs:

- an `MQueue` instance of any type (not empty)

```
>peek :: MQueue a → a
>peek q = (head.queueFront) q
```

A dequeue operation removes the element at the front of the queue. Most imperative implementations return the element removed, and modify the existing queue through a destructive update. Destructive updates are not possible in Haskell. Instead, one must create and return a new queue derived from the previous queue. To save time and storage space Haskell allows the new queue to maintain pointers to those parts of the original queue that were not modified, rather than create duplicate data (this is done automatically by Haskell to spare the programmer from maintaining pointer information). The `dequeue` function below takes a queue as input and returns the queue that results from removing the front element of the input queue. This action has the potential to empty the `queueFront` list and thus violate the invariant, so the `rebuildQueue` function (defined below) is used to fix the queue if such is the case. As with the `peek` function above, an error occurs if `dequeue` is called on an empty queue.

Inputs:

- an `MQueue` instance of any type (not empty)

```
>dequeue :: MQueue a → MQueue a
>dequeue q =
> rebuildQueue (q {queueFront = tail (queueFront q), queueSize = (queueSize q) - 1})
```

When one desires to see the abstract form of the queue, which is much simpler and easier to understand, then one can do so with the `listQueue` function. It returns the list made by concatenating the front list with the reverse of the rear list.

Inputs:

- an `MQueue` instance of any type

```
>listQueue :: MQueue a → [a]
>listQueue q = (queueFront q)++(reverse (queueRear q))
```

The `rebuildQueue` function checks and maintains the invariant that the front list can only be empty if the rear list is also empty. When it receives as input a queue with an empty front list it replaces the front list with the reversed rear list, and replaces the rear list with the empty list. The new queue represents the same abstract queue, but has a structure satisfying the invariant. Since none of the elements are lost in this process, the size remains the same. If `queueFront` is empty then the queue is returned unchanged, since it already satisfies the invariant.

Inputs:

- an `MQueue` instance of any type

```
>rebuildQueue :: MQueue a → MQueue a
>rebuildQueue (Q {queueFront = [], queueRear = r, queueSize = s})
>    = Q {queueFront = reverse r, queueRear = [], queueSize = s}
>rebuildQueue q = q
```

2.3.3 Binary Random Access List

Lists are the primary data structures in all functional programming languages and can often be used very effectively with recursion, which is the primary control mechanism in functional languages. However, the major weakness of linked lists in comparison with the arrays of imperative languages is that referencing elements by index (indexing) in lists takes linear time, whereas arrays can accomplish the same feat in constant time. The Binary Random Access List defined in this module provides a compromise between the two, in that all common list operations can be performed on it and it also allows for indexing in logarithmic time.

The Binary Random Access List (`BRAList`) presented in this module is a modified version of the one presented in Okasaki's book. The `BRAList` of this module is just one example of many different structures that can be designed based on the principle of binary number representation.

```
>module MyBRAList
> (BRAList,
> niceShowBRAList,
> isBRAListEmpty,
> consBRAList, headBRAList, tailBRAList,
> lengthBRAList,
> lookupBRAList, updateBRAList,
> listToBRAList, listFromBRAList,
> mapBRAList) where
```

The random access capabilities of a `BRAList` arise from its use of trees. Trees are common data structures in computer science, but they come in a variety of forms. Data structures involving trees very often have logarithmic access times because their structure is such that the number of elements under consideration at any given time is halved every time the focus of a search goes one level deeper in the tree (assuming the tree is balanced, which the trees in this module are).

Random access is implemented by storing a list of trees of increasing size. Each subsequent position in the list can hold a tree with twice as many elements as the tree in the previous index. The first tree in the list is a tree of a single element and has a size of one. The next tree in the list has two elements (elements are only stored at the leaves) and a size of two, and so on. All of these trees are perfectly balanced. In order to find a particular index in such a list, first the correct tree must be found and then the correct leaf node. The correct tree is found by subtracting tree sizes from the index value while searching through the list, until the index value is less than the size of the tree currently being examined. The correct position in the tree is found by searching to the left whenever the index is less than half the size of the tree and searching right otherwise. In either case, half the size of the tree is subtracted from the index before searching the next level.

The `Tree` data type can store any type of element. Every instance of `Tree` is either a `Leaf` or a `Node`. The `Leaf` instances store values and are at the bottom of every tree branch. The `Node` instances are junctions that store both a left tree and a right tree. All `Leaf` instances have a size of one, but a `Node` can have any size, depending on the size of its child trees. This size can be derived, but it is expedient to store it instead, since it is frequently used in indexing operations.

```
>data Tree a
> = Leaf {lValue::a}
> | Node {nSize::Int, leftT::Tree a, rightT::Tree a}
>    deriving (Show, Eq)
```

The `niceShowTree` function displays the elements within a tree in order from lowest index, left-most leaf, to highest index, right-most leaf, with hyphens separating values. It works as long as the values stored in the tree are showable (have a valid `Show` instance).

Inputs:

- a `Tree`

```
>niceShowTree ::  Show a ⇒ Tree a → String
>niceShowTree (Leaf x) = show x
>niceShowTree (Node _ t1 t2) = (niceShowTree t1)++‘‘-’’++(niceShowTree t2)
```

While storing elements in lists of trees allows for quick indexing, it makes common list operations very expensive. Adding or removing elements at the head of the list causes a cascade of updates to every tree, because every position in the list must contain a tree of a particular size and every tree must be balanced. The time complexity on all operations is logarithmic, because there can never be more than $\log n$ number of trees (where n is the number of elements). This is a dismal outcome compared with the constant time it takes to perform these operations on normal lists. To improve performance, a binary number representation of lists is applied. The worst case time is still logarithmic, but overall performance is increased.

All binary numbers can be represented by a list of two symbols (normally 0's and 1's). Traditional notation places the digit of smallest weight on the right, but for the purposes of the binary list representation the leftmost position will have the smallest weight instead. With such a representation for binary numbers one can see that incrementing a binary number is very similar to adding an element to the front of a list and decrementing a binary number is similar to taking the tail of a list. The difference is that binary arithmetic sometimes involves carrying and borrowing of digits from one place to the next.

A `BRAList` is made by assigning a binary number structure to a list of the `Tree` instances defined above. This is done by assigning the constructor `Zero` to the absence of a tree and the constructor `One` to the presence of a tree. The `One` constructor is simply a wrapper for `Tree` instances. These constructors come from the `Digit` data type defined below. Rather than simply being a list of trees, a `BRAList` is a list of `Digit` instances. This list is capable of storing `Zero` and `One` instances. The presence of `Zero` instances decreases the run time of most increment and decrement type operations. The worst case for an increment is a list of `One` instances, and the worst case for a decrement is a series of `Zero` instances preceding an instance of `One` (There are no lists containing only instances of `Zero`).

```
>data Digit a
> = Zero
> | One {cell::Tree a}
>    deriving (Show, Eq)
```

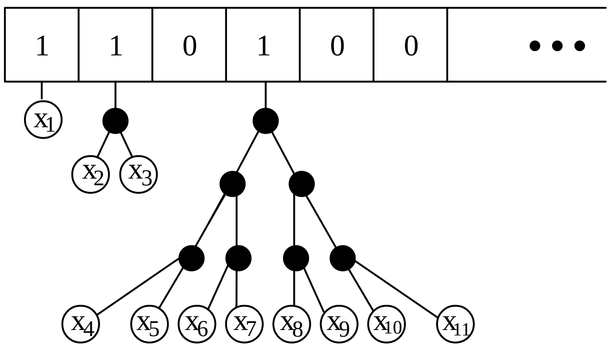
The `niceShowDigit` function displays blanks for an instance of `Zero` and the `Tree` instance for all instances of `One`.

Inputs:

- a `Digit` instance

```
>niceShowDigit ::  Show a ⇒ Digit a → String
>niceShowDigit Zero = ‘‘ ’’
>niceShowDigit (One t) = niceShowTree t
```

The `BRAList` type is a list of `Digit` instances. The size of the `Tree` instance allowed at each index of the `BRAList` is 2^i, where i is the index. Each index can have either a `One` or a `Zero` (with an infinite list of `Zero` instances assumed to be at the end of the list) and only the `One` instances store `Tree` instances.



An example `BRAList` with 11 elements: $x_1 \ldots x_{11}$. Positions in the list that hold a `Zero` type `Digit` hold no elements. `One` type `Digit`'s hold full, balanced `Tree`'s of elements.

```
>type BRAList a = [Digit a]
```

The `niceShowBRAList` function takes advantage of the `niceShowDigit` function, which it maps across the list. Because the `BRAList` is defined as a type synonym, it is still a regular list.

Inputs:

- a BRAList

```
>niceShowBRAList ::  Show a ⇒ BRAList a → String
>niceShowBRAList x = show (map niceShowDigit x)
```

The following set of functions that deal with the most basic operations on BRAList's are taken directly from the appendix of Okasaki's book. Only minor changes have been made for aesthetic purposes.

The sizeTree function returns the size of a Tree instance. All Leaf instances have a size of 1, and all Node instances track their size, making retrieval of the size very simple. The size of a tree is the number of elements it holds, which is the number of leaf nodes it has.

Inputs:

- a Tree

```
>sizeTree ::  Tree a → Int
>sizeTree (Leaf _ ) = 1
>sizeTree (Node w _ _ ) = w
```

Whenever an increment operation occurs, and the lowest position digit is not a Zero, carry over occurs at least once. The element being added is treated as a tree of one element: a leaf. Whenever a tree is added to an index that already has a tree in it, the two trees are linked into one tree of double the size and added to the next position in the list. This operation cascades until a tree is added to an instance of Zero, which is then replaced by an instance of One containing the newly linked tree. Linking two trees means making the first tree the left child of a new node and the other tree the right child of that node. The size of the resulting tree is obtained by adding the sizes of the two child trees.

Inputs:

- Tree of lower index elements
- Tree of higher index elements

```
>linkTree ::  Tree a → Tree a → Tree a
>linkTree t1 t2 = Node (sizeTree t1 + sizeTree t2) t1 t2
```

The consTree function is an abstraction of incrementing a binary number. The operation is very simple and occurs in constant time if either the initial BRAList is empty or if the first digit is a Zero instance. Otherwise the linkTree function from above is used to link the tree being added to the tree already at the index. That index is replaced with a Zero instance and the newly created tree is added to the remainder of the list with a recursive call to consTree. Such recursive calls continue to be made until one of the two base cases are reached, namely an empty list or Zero digit.

Inputs:

- Tree to add
- BRAList

```
>consTree ::  Tree a → BRAList a → BRAList a
>consTree t [] = [One t]
>consTree t (Zero:ts) = (One t):ts
>consTree t ((One t0):ts) = Zero:(consTree (linkTree t t0) ts)
```

The unconsTree function is an abstraction for decrementing a binary number, but it also does a little extra work. When decrementing an actual binary number it is always a value of 1 that is removed, but the One instances in BRAList's are different from one another in the contents of their trees. These contents are important and must be maintained and returned by the unconsTree function along with the result of removing that value from the BRAList. This time the simple cases are those in which the list starts with a One instance, because then that One simply becomes a Zero (or nothing if the list only had a single One in it). Its tree is the value returned. If the list starts with a Zero instance then a One is borrowed from the next available digit, which means that the left branch of the next tree is broken off and the right branch becomes the tree occupying the current index, which is one index lower than its former position. Of course, if the next index also holds a Zero instance then these borrows cascade until a One instance is found.

Inputs:

- BRAList

```
>unconsTree ::  BRAList a → (Tree a, BRAList a)
>unconsTree [] = error ‘‘Binary List is empty.  Cannot unconsTree.’’
>unconsTree [One t] = (t,[])
>unconsTree ((One t):ts) = (t,Zero:ts)
>unconsTree (Zero:ts) = (t1,(One t2):tss)
>    where
>       (Node _ t1 t2, tss) = unconsTree ts
```

Finding a particular index in a tree is easy because all of the trees are balanced and therefore hold an even number of elements. The first element in a tree, the leftmost leaf, has an index of 0. Therefore the left child of the tree's root contains elements whose indices are less than half the size of the tree, and the right child contains elements whose indices are greater than half the size of the tree. This holds true for all subtrees as well, so that when searching for a particular index, half of the remaining unsearched elements can be eliminated from the process at each step until there is only one element left. The only problem that could arise happens when an improper index is searched for, but since this function is only meant to be called by the lookupBRAList function, which has its own bound checking, an indexing error will not occur at this level.

Inputs:

- index at least zero and less than the size of the Tree
- Tree to search

```
>lookupTree ::  Int → Tree a → a
>lookupTree 0 (Leaf x) = x
>lookupTree i (Leaf _ ) = error ‘‘Index out of bounds on lookupTree.’’
>lookupTree i (Node w t1 t2)
> | i < halfS = lookupTree i t1
> | otherwise = lookupTree (i - halfS) t2
>    where
>       halfS = w div 2
```

The updateTree function searches for a given index in the same manner as the lookupTree function, but because it returns the updated tree instead of a single element within it, the function has to maintain information about the structure of the tree and then rebuild it as the successive recursive calls return their values. The tree is rebuilt from the bottom up, and the first function that returns a result returns a single leaf containing an updated element. This tree then replaces the tree previously at that position. The recursive calls keep returning until the root of the updated tree is returned.

Inputs:

- index at least zero and less than the size of the Tree
- value to replace the value at that index
- Tree to update

```
>updateTree :: Int → a → Tree a → Tree a
>updateTree 0 y (Leaf _ ) = Leaf y
>updateTree i _ (Leaf _ ) = error ‘‘Index out of bounds on updateTree.’’
>updateTree i y (Node w t1 t2)
> | i < halfS = Node w (updateTree i y t1) t2
> | otherwise = Node w t1 (updateTree (i - halfS) y t2)
>   where
>      halfS = w div 2
```

The isBRAListEmpty function checks to see if a BRAList is empty, and is nothing more than a call to null that is restricted to BRAList types.

Inputs:

- a BRAList

```
>isBRAListEmpty :: BRAList a → Bool
>isBRAListEmpty = null
```

The consBRAList function is a higher level function dealing with the incrementing abstraction. It is used to add a single element (rather than a tree as with consTree) to the front of a BRAList. To accomplish this the element is encapsulated in a Leaf instance and then added to the BRAList with consTree, which then takes over and deals with any necessary carry overs.

Inputs:

- element to add
- BRAList to add to

```
>consBRAList :: a → BRAList a → BRAList a
>consBRAList x ds = consTree (Leaf x) ds
```

The headBRAList function is very similar to the head function for normal lists. It returns the first element of a BRAList without changing the list itself. First it uses the decrementing abstraction unconsTree to get the Leaf instance containing the first element. The unconsTree function returns a 2-tuple containing this tree and the remaining BRAList, so the tree is taken from the tuple with the fst function (returns the first element of a 2-tuple). Then the lValue member function for Leaf instances is used to get the value contained within the leaf.

Inputs:

- a BRAList

```
>headBRAList :: BRAList a → a
>headBRAList ds = lValue (fst (unconsTree ds))
```

The tailBRAList function is much like the tail function for regular lists, and it represents the other half of the decrementing abstraction. Rather than retrieve the element removed during decrementing, this function returns the result of decrementing: the modified BRAList. It is very similar to the headBRAList list function and also makes use of unconsTree. The difference is that it retrieves the second value of the returned 2-tuple, with snd, rather than the first value.

Inputs:

- a BRAList

```
>tailBRAList :: BRAList a → BRAList a
>tailBRAList ds = snd (unconsTree ds)
```

The lookupTree function above finds a particular element in a given tree, but when searching for a particular index the correct tree must first be found by another function. This is accomplished by the lookupBRAList function. Given an index to search for, this function first finds the tree in which the index is located and then uses the lookupTree function to return the element at that index. The right tree is found by subtracting the size of the trees already searched from the index being searched for while going through the list. Naturally, Zero instances are ignored.

Inputs:

- index to search for
- BRAList to search

```
>lookupBRAList :: Int → BRAList a → a
>lookupBRAList i [] = error ‘‘Index out of bounds on lookupBRAList.’’
>lookupBRAList i (Zero:ds) = lookupBRAList i ds
>lookupBRAList i ((One t):ds)
> | i < sT = lookupTree i t
> | otherwise = lookupBRAList (i - sT) ds
>   where
>     sT = sizeTree t
```

The updateBRAList function searches through a BRAList in the same manner as the lookupBRAList function, but when it finds the tree it is looking for it uses the updateTree function instead of the lookupTree function. Since the entire updated BRAList needs to be returned, the function also maintains the structure of Digit instances that are passed over during the search process.

Inputs:

- index to search for
- value to replace the element at that index
- BRAList to search

```
>updateBRAList :: Int → a → BRAList a → BRAList a
>updateBRAList i y [] = error ‘‘Index out of bounds on updateBRAList.’’
>updateBRAList i y (Zero:ds) = Zero:(updateBRAList i y ds)
>updateBRAList i y ((One t):ds)
> | i < sT = (One (updateTree i y t)):ds
> | otherwise = (One t):(updateBRAList (i - sT) y ds)
>   where
>     sT = sizeTree t
```

The remaining functions in this module were not developed by Chris Okasaki.

The lengthBRAList function is much like the length function for regular lists. The length is calculated by ignoring instances of Zero and using the tree size when encountering instances of One. The length is the number of elements in the list, and the tree size is the number of elements in a given tree, so the length of the list is the sum of the sizes of all trees in the list.

Inputs:

- a BRAList

```
>lengthBRAList :: BRAList a → Int
>lengthBRAList [] = 0
>lengthBRAList (Zero:ds) = lengthBRAList ds
>lengthBRAList ((One t):ds) = (sizeTree t) + (lengthBRAList ds)
```

Because lists are so common in Haskell and there already exists a multitude of functions for them it is conceivable that one might switch between regular lists and `BRAList`'s in certain applications. The two functions below accomplish these tasks.

The `listToBRAList` function takes a regular list and converts it to a `BRAList` with repeated calls of the `consBRAList` function on the reverse of the original list.

Inputs:

- regular list

```
>listToBRAList :: [a] → BRAList a
>listToBRAList xs = ltoBRA (reverse xs) []
>    where
>       ltoBRA :: [a] → BRAList a → BRAList a
>       ltoBRA [] ds = ds
>       ltoBRA (x:xs) ds = ltoBRA xs (consBRAList x ds)
```

The `listFromBRAList` function accomplishes the opposite conversion with repeated calls of `headBRAList`. First on the initial `BRAList`, and then repeatedly of the results of `tailBRAList` for the first and each subsequently smaller list.

Inputs:

- a BRAList

```
>listFromBRAList :: BRAList a → [a]
>listFromBRAList [] = []
>listFromBRAList ds = (headBRAList ds):(listFromBRAList (tailBRAList ds))
```

Though the two functions above allow for switching back and forth between lists and `BRAList`'s, it is preferable to work exclusively with one data type. One of the most common and useful higher order functions for lists is `map`, and because it is so useful a version that works for `BRAList`'s is presented below. The `mapBRAList` function accomplishes the same task that a `map` would, namely to apply a given function to every element in a list and return the list of results. This is done by building a new list of results starting with the last element of the `BRAList`. This is done because as elements are added to the new list with `consBRAList`, they push back the values that are already in the list, so that at the completion of the function the ordering of the list of results corresponds to the list of inputs.

Inputs:

- function to map
- a BRAList

```
>mapBRAList :: (a → b) → BRAList a → BRAList b
>mapBRAList f ds = mapBRA (lengthBRAList ds - 1) []
> where
>    mapBRA (-1) bs = bs
>    mapBRA n bs = mapBRA (n - 1) (consBRAList (f (lookupBRAList n ds)) bs)
```

2.3.4 Binary Random Access Matrix

Whereas a Binary Random Access List (BRAList) is a functional answer to the one-dimensional arrays of imperative languages, the Binary Random Access Matrix (BRAMatrix) is the functional response to two-dimensional arrays. Although Haskell comes with a built in Array data type, it is inefficient when it comes to indexing operations. BRAList's are faster when performing indexing operations, but unlike the Array data type, only Int type indices are allowed, which makes all BRAList's one-dimensional. Strictly speaking, arrays of any dimension with any index type can be represented by Int indexed arrays (or BRAList's in this case) with the first index as zero because no matter how the cells are indexed, it is the number of cells that matter.

However, it can be easier to conceptualize and to code for data structures that more closely resemble the structure they represent. Indexing a one-dimensional array as if it were a two-dimensional array is a tedious task best left to assembly level programming. Because a higher level programming language such as Haskell allows for the definition of complex data structures, the programmer need not bother with such low level concerns. Haskell's Array type has such a higher level abstraction in that it allows its cells to be indexed with tuples (allowing for arrays of two or more dimensions), but as mentioned before, the Array type is inefficient. A new data type is needed.

A BRAMatrix is essentially a BRAList of BRAList's that can be indexed with 2-tuples. Several operations are defined on it which give it functionality similar to that of Haskell's Array type, but it also has the faster indexing speed provided by BRAList's. Unfortunately, BRAMatrix's lack the flexibility that BRAList's have. The BRAMatrix behaves much more like a traditional imperative two-dimensional array in that its size is defined at initialization and remains set through the matrix's lifetime.

```
>module MyBRAMatrix
> (BRAMatrix, matrixSize, grid,
> niceShowBRAMatrix,
> initialize2D,
> lookup2D, update2D,
> massUpdate2D, elems2D, assocs2D, accum2D) where
```

The BRAMatrix is composed of a BRAList of BRAList's, which is why code from the *Binary Random Access List* section is needed.

```
>import MyBRAList -- Binary Random Access List
```

A BRAMatrix must be initialized before it can be used. An uninitialized BRAMatrix is the constructor Null. After the BRAMatrix has been initialized, its size cannot be changed without causing inconsistencies. The initialized matrix has M as its constructor. The matrixSize data member is a 2-tuple of the length and width, in storage cells, of the matrix, and grid is the matrix itself.

```
>data BRAMatrix a
> = Null
> | M {matrixSize::(Int,Int), grid::BRAList (BRAList a)}
>    deriving (Show, Eq)
```

The niceShowBRAMatrix function displays a two-dimensional ASCII representation of the contents of the matrix. It uses listFromBRAList to obtain the contents of each individual row as a list. Then it prints each of these lists, separated by carriage returns, so that each subsequent row is printed under the previous row, making a table of data as output.

Inputs:

- a BRAMatrix

```
>niceShowBRAMatrix :: Show a ⇒ BRAMatrix a → String
>niceShowBRAMatrix (M {matrixSize = (_ , y), grid = g}) = showRows 0
>    where
>       showRows :: Int → String
>       showRows n
>          | n ≡ y = ‘‘’’
>          | otherwise
>            = (show (listFromBRAList (lookupBRAList n g)))++[‘\n’]++(showRows (n + 1))
```

The `initialize2D` function creates a new `BRAMatrix` of a given size with all cells initialized to a default value. Initialization of matrices is common in imperative languages, but the assignment of a default value to all cells is not. The cells of most matrices initialized in imperative languages start out containing random bits leftover in memory from previous operations. This type of uncertainty and machine state dependency is not allowed in Haskell. Therefore a default starting value for all cells must be given as input to the function.

The matrix is built by creating a repeating list of the default element and then transforming the list, containing the appropriate number of instances of the default element, into a `BRAList` with `listToBRAList`. This `BRAList` is then itself repeated, and the appropriate number of instances are taken from this repeating list to form the `grid` of `BRAMatrix` instance.

Inputs:

- 2-tuple of width and height of matrix
- default value that all cells are initialized to

```
>initialize2D :: (Int,Int) → a → BRAMatrix a
>initialize2D (x,y) d = M (x,y) (listToBRAList (take y (repeat (listToBRAList row))))
>    where
>       row = take x (repeat d)
```

Finding a specific index in a `BRAMatrix` is as simple as looking up the y-coordinate in the main list and the x-coordinate in the sublist, both times using `lookupBRAList`. The `grid` of the `BRAMatrix` is represented by a `BRAList` whose indices are y-coordinates. Each of these indices contains another `BRAList`, each of the same length, whose indices correspond to x-coordinates. The `grid` can be thought of as a list of the rows in the matrix. The `lookup2D` function uses the above method to find and return the value at a particular index.

Inputs:

- 2-tuple of xy index to look up
- BRAMatrix

```
>lookup2D :: (Int,Int) → BRAMatrix a → a
>lookup2D (x,y) m = lookupBRAList x (lookupBRAList y (grid m))
```

The `update2D` function finds the correct index in the same way as the `lookup2D` function, but it also returns the structure of the modified matrix. In the innermost parenthesis, the `BRAList` for the correct row is found, and then `updateBRAList` is used to update the proper x-coordinate within that list. The result is a modified `BRAList` for the entire row, which then replaces the row that used to be there, requiring another use of `updateBRAList`, this time with the y-coordinate. This modified `BRAList` of `BRAList`'s is then assigned to the `grid` of the matrix.

Inputs:

- 2-tuple of xy index to look up
- value to replace the element at the given index
- BRAMatrix

```
>update2D :: (Int,Int) → a → BRAMatrix a → BRAMatrix a
>update2D (x,y) n m
> = m {grid = updateBRAList y (updateBRAList x n (lookupBRAList y as)) as}
>    where
>       as = grid m
```

The remaining functions in this module are designed to provide much of the functionality of Haskell's `Array` type.

Often several changes need to be made to a matrix all at once rather than one at a time. Haskell's `Array` type provides a convenient `//` operator that accomplishes this operation by performing several updates according to a list of index-value pairs. The `massUpdate2D` function below works in much the same way. It recursively calls itself while calling `update2D` for each index-value pair in a list until the list is empty. The final matrix that is returned has had all of the indices in the input list updated to the corresponding values paired with those indices. It is important to note that the function makes no attempt to filter out duplicate indices. If the same index appears in the update list more than once then the last value in the list will overwrite any previous changes.

Inputs:

- BRAMatrix
- list of index-value pairs for updating the matrix

```
>massUpdate2D :: BRAMatrix a → [((Int,Int),a)] → BRAMatrix a
>massUpdate2D m [] = m
>massUpdate2D m ((p,n):us) = massUpdate2D (update2D p n m) us
```

Sometimes only the elements of a `BRAMatrix` are of interest, regardless of the elements' positions within the matrix's structure. The simplest container for a collection of elements is a generic list, and the `elems2D` function below transfers the elements of a `BRAMatrix` to just such a format. It works by performing `listFromBRAList` on each row of the `grid` and then appending the resulting lists together. The result is a list of all elements from the `BRAMatrix` independent of indices.

Inputs:

- BRAMatrix

```
>elems2D :: BRAMatrix a → [a]
>elems2D (M {matrixSize = (_ , y), grid = g}) = e2D 0
>    where
>       e2D :: Int → [a]
>       e2D n
>          | n ≡ y = []
>          | otherwise = (listFromBRAList (lookupBRAList n g))++(e2D (n + 1))
```

`BRAMatrix`'s provide both a structure and an indexing scheme that is useful for storing and accessing elements quickly. The `elems2D` function removes both the structure and the indexing scheme from the elements, which is useful when neither of these features matter. However, sometimes the indexing scheme of a matrix is needed, but the structure is undesirable. This is mainly because there already exists a wide variety of functions for handling regular lists, and to attempt remaking all of these functions for `BRAMatrix`'s would be impractical. The `assocs2D` function strips the structure of a `BRAMatrix` away from its elements by putting them in a list, but it maintains the indexing scheme by pairing each value with its position in the matrix. This function relies partly on the `elems2D` function and takes advantage of the fact that even though the values in the list generated by `elems2D` have no indices, they are still ordered according to their positions in the original `BRAMatrix`. Therefore the `assocs2D` function uses `elems2D` to strip all elements from a matrix and then uses `zip` to recombine these elements with the appropriate indices, generated by a list comprehension.

Inputs:

- `BRAMatrix`

```
>assocs2D ::  BRAMatrix a → [((Int,Int),a)]
>assocs2D m = zip [(x,y) | y ← [ 0,...,s_y − 1 ], x ← [ 0,...,s_x − 1 ]] (elems2D m)
>    where
>       (s_x, s_y) = matrixSize m
```

The `massUpdate2D` function performs multiple replacement updates on elements of a `BRAMatrix`. Sometimes a modification to the existing value is preferred over its replacement, which is what the `accum2D` function is used for. The `accum2D` function performs a mass update on a `BRAMatrix` according to a list of index-value pairs, but instead of replacing each of the indices in the list it modifies them according to a function provided as input. The `accum2D` function is therefore a higher order function. The function given as input must take a value from the matrix and a value of another or same type, and produce a new value that replaces the previous value in the matrix at the given index. These changes are accumulative, hence the name `accum2D`, so that if the same index appears in the index-value list more than once, then that particular index will be modified more than once. The series of changes build upon one another rather than replacing each other. Then after all the changes are made the new matrix is returned.

Inputs:

- function taking an element and another value to produce a new element
- `BRAMatrix`
- list of index-value pairs for updating the matrix

```
>accum2D ::  (a → b → a) → BRAMatrix a → [((Int,Int),b)] → BRAMatrix a
>accum2D f m [] = m
>accum2D f m ((p,u):cs) = accum2D f (update2D p (f (lookup2D p m) u) m) cs
```

2.4 Neural Networks

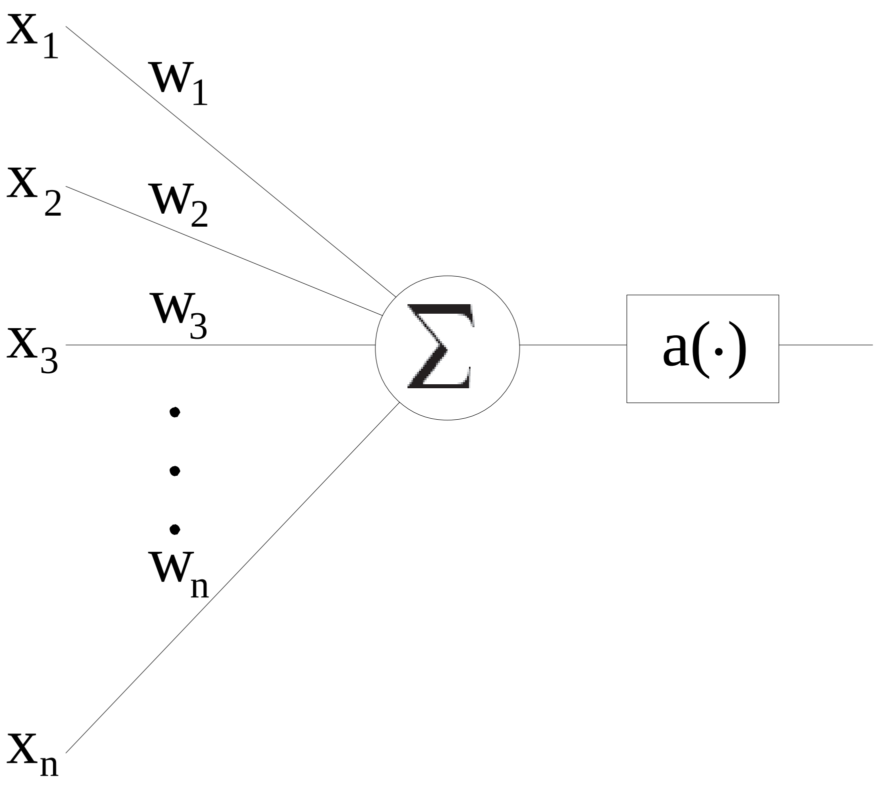
2.4.1 Overview

Artificial neural networks are tools of artificial intelligence based on the workings of the brain. They come in a variety of forms, one of which is used in this simulation. The neural networks used in this simulation support Hebbian learning and discrete time delays. The first module of this section develops these neural networks, and the second module contains activation functions, which are used by neural networks to determine the outputs of neural synapses.

2.4.2 Neural Networks Supporting Discrete Time Delays and Hebbian Learning

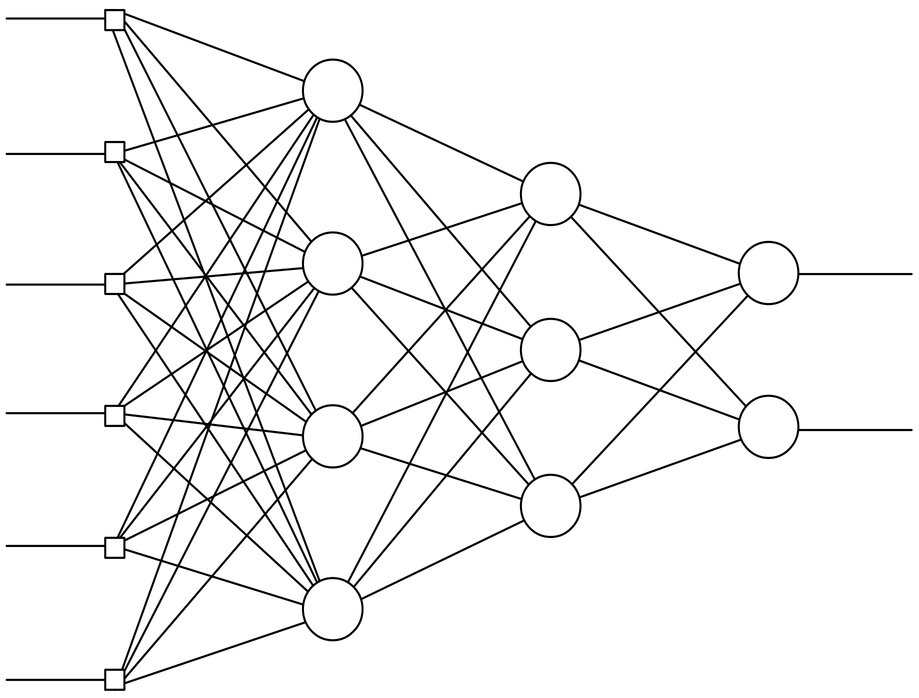
The theory behind neural networks is based on the structure and learning processes of the brain. Within the brain are cells called neurons. Several tendrils, called dendrites, extend from each neuron. Each neuron has a single long tendril called an axon. The axons of neurons extend towards the dendrites of other neurons. The two are separated from each other by small spaces called synapses. Each neuron is capable of transmitting an electrical signal through its axon. This signal is transmitted from the end of the axon to the dendrites of other neurons through the synapses between them, though the strength of the connection varies from synapse to synapse. The many dendrites of a neuron gather such electrical impulses and carry them to the neuron. Depending on the nature of the electrical signal, these impulses can be either excitory of inhibitory. If the neuron becomes excited enough, then it will transmit a signal along its axon, and so it goes from neuron to neuron.

Neural networks simulate this process with artificial neurons and synapses. A neural network can be a physical construct or a software construct, such as what it presented in this module. The software neurons in the module receive floating point number inputs instead of electrical signals. These inputs are weighted and the sum of these weighted values is sent as input to an activation function. The activation function essentially determines if the neuron is excited enough to send a signal. Neurons in the brain either do or do not send a signal depending on how excited they are. Neural network neurons can also behave this way, if a *threshold* activation function is used, but they need not be so limited. An activation function is any function that takes a floating point number as input and returns a floating point number is output (normally in the range [0,1]).



A single artificial neuron. Each x_i is an input along a synapse with synaptic weight w_i. The synapses flow into the summing junction where $\sum_{i=1}^{n} x_i w_i$ is calculated. This result is passed to the activation function $a(\cdot)$, and the output from this function is the output from the neuron.

Neural network neurons are not restricted to having only one artificial axon. They can transmit a signal across several pathways simultaneously. There are many different ways to define the architecture of neural networks, but the networks that can be defined within this module share a standardized set of features. Each neural network is composed of several layers of artificial neurons, henceforth simply called nodes. Synapses only exist between nodes in adjacent layers. The networks defined in this module are fully connected, meaning that every node in one layer is connected to every node in adjacent layers. Every neural network has at least a set of input pathways and an output layer. If the network only has one layer then that layer is both the input layer and the output layer. If the network has more layers, then the layer into which the initial inputs flow is the input layer, and all layers between this layer and the output layer are called hidden layers.


An example fully connected neural network with 6 inputs and 2 outputs. Signals flow from left to right.

Each node in the network receives input from all nodes of the previous layer (or from the initial inputs), calculates a corresponding output signal, and then transmits this result to all nodes of the next layer. In order to simulate the varying strengths of synaptic connections in the brain, each artificial synapse has a weight associated with it. The signal that is transmitted by the synapse equals its input multiplied by the synaptic weight.

These neural networks also allow for the possibility of discrete time delays along individual synapses. Since these neural networks are intended for use in an environment that changes throughout time and every set of inputs provides information about a particular moment, such time delays provide an abstract form of short term memory. A discrete time delay is a number of time steps for which a synapse withholds its signal. The first time a time delayed synapse receives input it stores that input and transmits a zero as output. It keeps storing inputs until its time delay value is reached, which means that a number of discrete moments equal to the time delay value have passed. At this point it transmits the first input value it received as output. This value is then multiplied by the synaptic weight. This means that the neural network has at least partial access to information from the past. With a mixture of time delayed and continuous synapses a neural network is able to both be aware of what is currently happening and what happened in the recent past. These time delays are implemented with queues that enqueue new input values and dequeue old input values.

Yet another special feature of these neural networks is their support for Hebbian learning. Hebbian Learning is based on the work of neuropsychologist Donald O. Hebb. Hebb's Postulate, which he presents in his book "The Organization of Behaviour", is the following:

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased. [2]

This means that synaptic connections become stronger when signals transmitted along them result in more signals being transmitted. This allows neural networks to learn, and is as such a form of long term memory. In a neural network, learning means changing the weight along a synapse in question, but the criteria for when and how must be defined. If a threshold activation function is used then the synaptic weight is increased whenever the firing of one node contributes to the firing of another, as stated in Hebb's Postulate, but since this module also supports other types of activation functions, the mechanism for the modification of synaptic weights must be further developed.

The change Δw_{ij} in the weight of the synapse from node i to node j on each time step is given by the following equation [1]:

$$\Delta w_{ij} = n(x_i - x'_i)(y_j - y'_j)$$

where
n = Hebbian learning rate parameter
x'_i = the average value of all past inputs to node j (outputs from node i)
x_i = current input to node j (output from node i)
y'_j = the average value of all past outputs from node j
y_j = current output from node j

The Hebbian learning rate parameter is a constant that affects the rate of learning. It is normally a small number (less than one, greater than zero), and the larger it is, the faster learning occurs, or more accurately, the modifications to the synaptic weights are larger. The reason that the average values of previous inputs and outputs are used is to prevent saturation. If the input and output values were used without subtracting the previous averages, then all synaptic weights would quickly approach infinity or negative infinity.

The advantages of using Hebbian learning are that it is an unsupervised learning method and it is based on real-world biology. Since the neural networks of this module are used within independent artificial organisms, it would make no sense, and be extremely difficult if not impossible, to apply a supervised learning algorithm. The organisms have to be able to function without outside supervision. Since Hebbian learning is based on observations of brain activity in real living organisms, it makes sense to apply the same mechanism in artificial organisms.

```
>module HebbTimeNet
> (module MyQueue,
> Synapse, Node, Layer, Net,
> calcNet,
> netFromLists,
> getNodeList, countSynapses) where
```

Queues are used to implement discrete time delays along neural network synapses.

```
>import MyQueue -- Queue
```

A synapse within a neural network is a connection between two nodes. Every synapse carries a synaptic weight representing the strength of the connection between the two nodes. A synapse with a negative weight sends an inhibitory signal and one with a positive weight sends an excitory signal. The closer the weight is to zero the weaker the connection is, and the farther from zero, the stronger the connection.

Since these synapses support Hebbian learning, each one needs to have a way of determining the average of all of its past inputs. This value is the sum of all previous inputs divided by the number of input sets received. In order to calculate this, each individual synapse remembers the sum of all previous inputs with the `totInput` data member.

The `Synapse` data type supports two constructors. The `Continuous` constructor is for synapses that do not have time delays (or rather, have a time delay of zero). They pass on every signal that they receive as soon as they get it. The `Delayed` synapse is for synapses with a discrete time delay. The value of the delay must be a positive `Int` value, and is stored in the `delay` data member. The input values that have been delayed by the synapse, and are currently waiting to be transmitted, are stored in the queue named `waiting`. Whatever value is at the front of this queue is the next one to be transmitted.

```
>data Synapse
> = Continuous {weight::Float, totInput::Float}
> | Delayed {weight::Float, totInput::Float, delay::Int, waiting::(MQueue Float)}
>    deriving (Eq, Show)
```

Every node receives input signals from several different sources and sends only one output signal, albeit along several pathways. That means that a node is little more than the collection of synapses that send the node input. Every instance of the `Node` data type is created with the `Junction` constructor, and the collection of `Synapse` data objects that input into it is a list called `inputs`. Each node also maintains the sum of all previous outputs in the `totOutput` data member. This value is needed to derive the average of all past inputs for use by the Hebbian learning equation.

```
>data Node = Junction {inputs::[Synapse], totOutput::Float}
>    deriving (Eq, Show)
```

A layer of nodes is so simple that it is represented with a type synonym rather than a data type. A `Layer` is a list of nodes, all of which are connected to each node of the previous layer. This means that every node in a given layer has the same number of synapses within its `inputs` member, so that the neural network is consistent.

```
>type Layer = [Node]
```

The neural network is represented by the `Net` data type. It contains a data member called `net` which is a list of the layers in the neural network from the input layer to the output layer. The structure of `net` is recursive in that its `tail` also represents a neural network. The functions below take advantage of this fact when calculating the result from passing inputs through the network. There is also a data member called `step` which stands for "time step" and tracks the number of previous input sets received by the neural network. This value is used to calculate the average input and output values of synapses when applying Hebbian learning.

```
>data Net = Neural {net::[Layer], step::Integer}
>    deriving (Eq, Show)
```

Every synapse receives input signals and produces output signals. The output signal is equal to the input signal multiplied by the synaptic weight. However, only continuous synapses are this simple. If the synapse is time delayed, then the input signal is instead enqueued into the `waiting` queue, and the value dequeued from this queue is multiplied by the synaptic weight to get the output value instead. This means that the *effective* input to a time delayed synapse is the value that it dequeues from the `waiting` queue. The operations listed above are performed by the `calcSynapse` function below. This function returns as output a 3-tuple of different values. The first of these values is the *effective* input to the synapse, which is important for Hebbian learning. The second value is the now updated synapse. This has to be done because Haskell does not allow for destructive updates on data structures. The third value of the 3-tuple is the output value calculated by the synapse.

Inputs:

- synaptic input
- synapse

```
>calcSynapse ::  Float → Synapse → (Float, Synapse, Float)
>calcSynapse i (Continuous w ti) = (i, Continuous w ti, w × i)
>calcSynapse i d
> | queueSize q < delay d = (0, d {waiting = enqueue q i}, 0)
> | otherwise = (signal, d {waiting = enqueue (dequeue q) i}, signal × weight d)
>    where
>       q = waiting d
>       signal = peek q
```

Hebbian learning is modeled by the `learnSynapse` function below. After the output of a synapse has already been calculated, the `learnSynapse` function is used to determine how the weight of the synapse is adjusted. Several pieces of input are needed. The Hebbian learning rate parameter is a constant. The current time step (number of previous input sets) is taken from the `Net` data instance of the neural network. The average of all past outputs from the node that the synapse connects to is received by this function as input. The current output and current effective input to the synapse are returned by the `calcSynapse` function above. The Hebbian learning equation is applied within the `where` clause of the function below.

Inputs:

- learning rate parameter
- current time step
- average of past outputs of node
- current output of node
- current effective input to synapse
- synapse

```
>learnSynapse ::  Float → Integer → Float → Float → Float → Synapse → Synapse
>learnSynapse n s y′ y x syn = syn {weight = w + Δw, totInput = i + x }
>    where
>       w = weight syn
>       i = totInput syn
>       Δw = n (x - x′)(y - y′)
>       steps = fromInteger s
>       x′ =
>          if steps ≡ 0 then 0
>          else (i / steps)
```

A node receives a list of inputs and creates one output. This output is calculated by summing the weighted inputs from each synapse and sending the result to an activation function. The activation function is one of the inputs to the `calcNode` function below. The length of the list of inputs sent to the `calcNode` function must equal the number of synapses in the node's `inputs` list in order to be consistent. The output of this function is a 2-tuple containing the output from the node and the updated version of the node. The node is updated because its synapses change as a result of using both the `learnSynapse` and `calcSynapse` functions. The average of previous outputs of the node, which is needed by `learnSynapse`, is calculated within this function by dividing the `totOutput` member of the `Node` instance by the current time step taken from the `Net` that the node is a member of.

Inputs:

- activation function
- learning rate parameter
- current time step
- list of inputs
- node

```
>calcNode :: (Float → Float) → Float → Integer → [Float] → Node → (Node,Float)
>calcNode a n s is node = (lss, output)
>    where
>       output = a (sum os)
>       (tis, nss, os) = unzip3 (zipWith calcSynapse is (inputs node))
>       lss = Junction
>          {inputs = zipWith (learnSynapse n s y' output) tis nss,
>          totOutput = totalOut + output}
>       totalOut = totOutput node
>       y' =
>          if s ≡ 0 then 0
>          else (totalOut / (fromInteger s))
```

Because layers are fully connected, every node within one layer receives the exact same set of unweighted inputs. Since the inputs are constant across the layer, calculating the list of outputs from a layer can be done by mapping `calcNode` across the list of nodes in the layer. This returns a list of 2-tuples, each containing an updated node and its output value.

Inputs:

- activation function
- learning rate parameter
- current time step
- list of inputs
- layer/list of nodes

```
>calcLayer :: (Float → Float) → Float → Integer → [Float] → Layer → [(Node,Float)]
>calcLayer a n s is = map (calcNode a n s is)
```

The end output from the neural network is calculated by the `calcNet` function, which makes recursive calls to itself and successively applies the `calcLayer` function. The function first sends the initial inputs to the input layer of the neural network and calculates the outputs of that layer. That layer has now been updated, but for the moment it is ignored as the `calcNet` function determines the final output of the neural network. It does this by stripping the lowest layer from the neural network and treating the remaining layers as a neural network itself. The outputs from the previous layer are used as the inputs to this reduced neural network. Recursion continues to strip away the layers of the network until there are none left, and the final output values have been calculated. The output of the function is these output values along with the updated neural network, which is reconstructed by the `join` helper function as the previous series of recursive calls unwind one after the other.

Inputs:

- activation function
- learning rate parameter
- list of inputs
- neural network

```
>calcNet :: (Float → Float) → Float → [Float] → Net → (Net,[Float])
>calcNet a n is (Neural [] s) = (Neural [] (s+1), is)
>calcNet a n is (Neural (l:ls) s) = (join nl nn, os)
>   where
>      (nl, nis) = unzip (calcLayer a n s is l)
>      (nn, os) = calcNet a n nis (Neural ls s)

>      join :: Layer → Net → Net
>      join l n2 = n2 {net = l:(net n2)}
```

The above functions all deal with sending inputs to and calculating outputs from a neural network, but they only work when applied to a properly constructed neural network. To create a neural network using the constructors provided above is tedious, so the following functions both make the task easier and assure the correctness of the resulting neural network.

The `makeSynapse` function creates a brand new synapse with its `totInput` field set to 0, since it has yet to receive any input. The two inputs to the function are the time delay and the weight for the synapse. If the time delay is 0 then the `Continuous` constructor is used and otherwise the `Delayed` constructor is used. The time delay must be at least zero.

Inputs:

- time delay
- synaptic weight

```
>makeSynapse :: Int → Float → Synapse
>makeSynapse 0 w = Continuous {weight = w, totInput = 0}
>makeSynapse d w = Delayed {weight = w, totInput = 0, delay = d, waiting = createQueue []}
```

Making a new node is as simple as calling the `makeSynapse` function repeatedly with different inputs. The `makeNode` function does this with a `zipWith` call and stores the result in its `inputs` member. The `totOutput` is set to 0 since the node has yet to create any output. The list of time delays must be the same length as the list of synaptic weights.

Inputs:

- list of time delays
- list of synaptic weights

```
>makeNode :: [Int] → [Float] → Node
>makeNode ds ws = Junction {inputs = zipWith makeSynapse ds ws, totOutput = 0}
```

Once again taking advantage of the `zipWith` function, the `makeLayer` function creates a layer by calling the `makeNode` function repeatedly.

Inputs:

- list of lists where each list contains the time delays for one node
- list of lists where each list contains the synaptic weights for one node

```
>makeLayer :: [[Int]] → [[Float]] → Layer
>makeLayer = zipWith makeNode
```

The makeNet function also takes advantage of the zipWith function, this time using the makeLayer function. The step member of the Net instance is set to 0 because the network has yet to receive input.

Inputs:

- list of lists of lists of time delays grouped first by node and then by layer
- list of lists of lists of synaptic weights grouped first by node and then by layer

```
>makeNet ::  [[[Int]]] → [[[Float]]] → Net
>makeNet ds ws = Neural {net = zipWith makeLayer ds ws, step = 0}
```

Even this simplified function has messy and complicated input, so one more function is defined to further simplify the creation of neural networks. The netFromLists function takes a single list of time delays and a single list of synaptic weights as input, instead of lists within lists within lists. Information about how to break up the delays and weights is lost when the data is represented in this form, so another input is needed. This input is a list of the number of nodes per layer in the neural network. Technically, the first number in this list is the number of input sources rather than a number of nodes. Therefore this list has one more value than the number of layers in the network. By looking at any two adjacent values in this list the number of synapses between each layer of the network can be calculated by multiplying the values together. The function below calculates this product for each pair of values and pulls this number of weights and time delays from their respective lists in order to create each layer.

Inputs:

- list of nodes per layer (starting with number of input sources)
- list of time delays
- list of synaptic weights

```
>netFromLists ::  [Int] → [Int] → [Float] → Net
>netFromLists nn dd ww = makeNet dds wws
>    where
>       (dds, wws) = (makeN nn dd, makeN nn ww)

>       makeN ::  [Int] → [a] → [[[a]]]
>       makeN (n:[]) _ = []
>       makeN (n1:n2:ns) xs
>          = (makeL n1 n2 (take val xs)):(makeN (n2:ns) (drop val xs))
>             where val = n1 × n2

>       makeL ::  Int → Int → [a] → [[a]]
>       makeL _ _ [] = []
>       makeL _ 0 _ = []
>       makeL n1 n2 xs = (take n1 xs):(makeL n1 (n2 − 1) (drop n1 xs))
```

The getNodeList function takes a neural network as input and derives the corresponding list of nodes per layer by retrieving the length of each layer. The first value, number of input sources, is found by counting the number of inputs to a node of the first layer.

Inputs:

- a neural network

```
>getNodeList :: Net → [Int]
>getNodeList n = ((length.inputs.head.head) nn):(gnl nn)
>    where
>        nn = net n

>        gnl :: [Layer] → [Int]
>        gnl [] = []
>        gnl (g:gs) = (length g):(gnl gs)
```

Once one has the node list, representing the architecture of a neural network, one can use it to determine the total number of synapses in the neural network. The `countSynapses` function takes a list of nodes per layer and uses it to count the synapses in the network. It does this by summing the products of adjacent values in the list. The product of adjacent values counts the number of synapses between two layers because the network is fully connected.

Inputs:

- a node per layer list (network architecture)

```
>countSynapses :: [Int] → Int
>countSynapses (b:[]) = 0
>countSynapses (a:b:cs) = (a × b) + (countSynapses (b:cs))
```

2.4.3 Neural Network Activation Functions

The activation function of a neural network maps the sum of inputs into a single node to the output of that node. At every junction of a neural network the input from each node of the previous layer is summed. This sum is the input to the activation function. This value is then transmitted from the node to all nodes of the next layer.

The three types of activation functions presented below are threshold, piecewise and sigmoid. For each function there is a so-called “positive” version and a “full” version. The positive versions map to the range [0,1] and the full versions map to the range [-1,1].

```
>module ActFunctions
> (actThresholdPos, actThresholdFull,
> actPiecewisePos, actPiecewiseFull,
> actSigmoidPos, actSigmoidFull) where
```

Threshold activation functions are simple functions that convert all input to one of several discrete values. The positive version maps to 0 or 1, and the full version maps to -1, 0 or 1.

Positive Threshold Activation Function
Range: 0 or 1
Inputs:

- floating point number

```
>actThresholdPos :: (Floating a, Ord a) ⇒ a → a
>actThresholdPos x
> | x > 0 = 1
> | otherwise = 0
```

Full Threshold Activation Function
Range: -1, 0 or 1
Inputs:

- floating point number

```
>actThresholdFull ::  (Floating a, Ord a) ⇒ a → a
>actThresholdFull x
> | x > 0 = 1
> | x ≡ 0 = 0
> | otherwise = −1
```

Piecewise functions behave like threshold functions for large values, but for smaller values in a given range the input matches the output. For the positive version this range is [0,1], and for the full version this range is [-1,1].

Positive Piecewise Activation Function
Range: [0, 1]
Inputs:

- floating point number

```
>actPiecewisePos ::  (Floating a, Ord a) ⇒ a → a
>actPiecewisePos x
> | x ≥ 1 = 1
> | x ≤ 0 = 0
> | otherwise = x
```

Full Piecewise Activation Function
Range: [-1, 1]
Inputs:

- floating point number

```
>actPiecewiseFull ::  (Floating a, Ord a) ⇒ a → a
>actPiecewiseFull x
> | x ≥ 1 = 1
> | x ≤ −1 = −1
> | otherwise = x
```

Sigmoid functions produce smooth curves whose outputs approach either a lower or upper asymptote as the input approaches negative or positive infinity respectively. In pure form, the output never reaches the values of the asymptotes, but because the computer has only finite storage it cannot properly represent the difference between the actual value and the value of the asymptote. This means that the outputs of very small and very large inputs are rounded to the value of the asymptote being approached. This is the case with the positive sigmoid function. The full sigmoid function has an additional problem. It is represented by the hyperbolic tangent function, which conveniently has the properties of a sigmoid function and has a range of (-1,1). However, the Haskell `tanh` function does not return proper output on large positive or negative inputs. Therefore the full sigmoid function is represented by a piecewise function, with the center piece being `tanh`. The pieces are cut at inputs of -88 and 88 because beyond these points the `tanh` function rounds output to -1 and 1 respectively.

Positive Sigmoid Activation Function
Range: [0, 1]
Inputs:

- floating point number

```
>actSigmoidPos :: (Floating a, Ord a) ⇒ a → a
>actSigmoidPos x = 1/(1 + e^{-x})
```

Full Sigmoid Activation Function
Range: [-1, 1]
Inputs:

- floating point number

```
>actSigmoidFull :: (Floating a, Ord a) ⇒ a → a
>actSigmoidFull x
> | x > 88 = 1
> | x < −88 = −1
> | otherwise = tanh x
```

2.5 World: Part 1

2.5.1 Overview

This section contains the *Environment* and *Constants* modules. They start the process of defining the environment, which is completed in the *Bins* and *Simulation* sections. However, these sections are impossible to understand without first discussing plants and animals, which are covered in the sections *Plants* and *Animals* respectively. By the same token, those sections make no sense without first reading the *Environment* and *Constants* sections. Therefore the section on the world is divided into two parts.

World: Part 1 defines the basic underlying structure of the world in *Environment*, and several important constants in *Constants*. Changing the values of these constants is how different data sets are generated, so the values assigned to the constants within this section should only be thought of as possibilities, and not as final values. The details of which constants were used to generate which set of data are explained later in the *Results* section.

World: Part 2 contains *Bins* and *Simulation*, which complete the definition of the world in which the simulation occurs.

2.5.2 Environment

The functions of this module are very general in their applicability, but they all deal with the mechanics and geometry of the environment within which the simulation takes place. The environment is a rectangle within the first quadrant of the two-dimensional (2D) coordinate plane. This means that all valid coordinates are positive for both x and y. The environment wraps around to the opposite side in both the vertical and horizontal directions, meaning that the environment can be envisioned as the surface of a torus, which is a mathematical term for a donut shaped object. Like the surface of a torus, the environment has no edges. Positions are referred to by their planar coordinates. The purpose of some of the functions below is to resolve the problems that arise from having this dual representation. Other functions deal with the way animals perceive the environment, while yet more deal with simple geometry issues. This collection of functions is diverse, but related in that they all deal with how the environment is modeled.

An important reccurring feature of the environment is 2D arrays, which are implemented with `BRAMatrix`'s. Arrays are used in conjunction with `Bin`'s, which are explained in full detail in the section titled *Bins*. However, some knowledge of `Bin`'s is required in order to understand the layout of the environment. As mentioned above, the world is rectangular in shape. This rectangle is further divided into smaller rectangles of equal size, each of which is represented by a `Bin`. When an animal wanders off the edge of one `Bin` it enters the adjacent `Bin` at the opposite edge. All `Bin`'s are stored in a 2D array, and two `Bin`'s are considered adjacent if they occupy adjacent cells within the array. Due to the torus-like nature of the environment, `Bin`'s on the edge of this array are also considered adjacent to those on the opposite side of the array. The 2D arrays are used to define `PlantGrid`'s, which are defined in the *Plants* section. Every `Bin` contains its own `PlantGrid`, which is superimposed upon the space contained therein, dividing it into yet smaller regions in which plants can grow. Each cell within a `PlantGrid` can support a single plant, and a `PlantGrid` maintains information for whether or not a plant occupies a given space.

```
>module Environment
> (confineArrayIndex,
> checkCoordBounds, wrapCoordinates, distance,
> relativeQuadrant, confineHeading) where
```

A 2D array has cells referenced by x and y coordinates like the coordinate plane of the environment. The main difference in these coordinate sets is that all array cells are referenced with `Int` values. As with the overall environment, array positions can wrap around the vertical axis and the horizontal axis. Since all array coordinates are made up of `Int` values, this can be easily accomplished using modulus division by the size of the array along the x axis and y axis.

Inputs:

- 2-tuple of the width and height of the array
- a 2D index which may or may not be within the boundaries of the array

```
>confineArrayIndex ::  (Int,Int) → (Int,Int) → (Int,Int)
>confineArrayIndex (w,h) (x,y) = (x mod w, y mod h)
```

When a point is outside the boundaries of a given region, rather than immediately translating the point to a new position by wrapping it around, it is useful to know which neighboring region the point occupies in relation to the given region. Given that the boundaries define a rectangular region and that the origin to which all points are related is the lower left corner of the rectangle, the region that the point occupies can be determined with a few tests. Eight regions are defined around the bounded region such that the bounded region is like the center tile in a tic-tac-toe board. There are regions to the left, right, top and bottom as well as the upper-left, upper-right, lower-left and lower-right. Each of these regions is assigned an `Int` value: lower-left is 1, left is 2, upper-left is 3, top is 4, upper-right is 5, right is 6, lower-right is 7 and the bottom is 8. The regions are numbered in order around the bounded region in the clockwise direction (see table). To determine which region a point lies in, each coordinate is compared to both 0 and the maximum possible value of the coordinate, which is given as input. If the point lies inside the boundaries of the region then a 0 is returned.

3	4	5
2	0	6
1	8	7

Numbers assigned to regions centered about the 0-region by `checkCoordBounds`.

Inputs:

- 2-tuple of the maximum coordinates of a 2D region (ordered numeric type)
- point which may or may not be within the boundaries of the region

```
>checkCoordBounds ::  (Ord a, Num a) ⇒ (a, a) → (a, a) → Int
>checkCoordBounds ($m_x$, $m_y$) (x,y)
> | not (left ∨ right ∨ top ∨ down) = 0
> | left ∧ down = 1
> | left ∧ top = 3
> | left = 2
> | right ∧ down = 7
> | right ∧ top = 5
> | right = 6
> | top = 4
> | down = 8
> where
>    left = x < 0
>    right = x ≥ $m_x$
>    down = y < 0
>    top = y ≥ $m_y$
```

The next function works for any ordered numeric type, but it is essentially the floating point equivalent of the `confineArrayIndex` function above. Like `confineArrayIndex`, the `wrapCoordinates` function assures that a point is within the boundaries of a region, but with floating point numbers one cannot use modulus division. The `wrapCoordinates` function works instead by adding or subtracting the distance along the corresponding dimension to or from the value of the coordinate that is out of bounds. This also works for integral types, but is unnecessary because modulus division is equivalent. The function keeps calling itself, modifying no more than one coordinate at a time until both coordinate values are within bounds. It makes a maximum of two recursive calls. The value that is within bounds is returned.

Inputs:

- 2-tuple of the maximum coordinates of a 2D region (ordered numeric type)
- point which may or may not be within the boundaries of the region

```
>wrapCoordinates ::  (Ord a, Num a) ⇒ (a, a) → (a, a) → (a, a)
>wrapCoordinates ($m_x$, $m_y$) (x,y)
> | x < 0 = wrapCoordinates ($m_x$, $m_y$) ($m_x$ + x, y)
> | y < 0 = wrapCoordinates ($m_x$, $m_y$) (x, $m_y$ + y)
> | x ≥ $m_x$ = wrapCoordinates ($m_x$, $m_y$) (x - $m_x$, y)
> | y ≥ $m_y$ = wrapCoordinates ($m_x$, $m_y$) (x, y - $m_y$)
> | otherwise = (x,y)
```

One of the most basic pieces of information that can be derived when one has two points is the distance between them. Since all of the 2D coordinates are rectangular coordinates, the distances between coordinates along like dimensions represents the lengths of legs of a right triangle, in which case the length of the hypotenuse is the distance between the two points. The length of the hypotenuse of a right triangle is easily found using the Pythagorean Theorem, stating that the square of the hypotenuse is equal to the sum of the squares of each leg. Because square root is used in the function below, the coordinates must be of a floating point type.

Inputs:

- first rectangular coordinate
- second rectangular coordinate

```
>distance ::  Floating a ⇒ (a, a) → (a, a) → a
>distance ($x_1$, $y_1$) ($x_2$, $y_2$) = $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
```

Animals are self-centered entities. They perceive the world relative to their current position and orientation/heading. Therefore the `transform` function is defined below to translate the position of one point to a point relative to the position and heading of another. The first point, called the focus point, has an xy position and a heading. In the relative coordinate system this point is the origin, so the first step in the transformation is to subtract the x and y coordinates of the focus point from the x and y coordinates of the relative point. The next step is to rotate the coordinate plane to align the heading of the focus point with the positive x axis. This is accomplished with the rotation matrix shown below.

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Rotation matrix

The vector resulting from the matrix multiplication above is the position of the relative point after having been rotated θ radians. It should be noted that the xy vector multiplied by the matrix above has already gone through the first step of translation, namely subtraction of the focus point's x and y values.

Inputs:

- position of focus point
- orientation/heading of focus point
- position of relative point

```
>transform ::  (Floating a, Ord a) ⇒ (a, a) → a → (a, a) → (a, a)
>transform (xf,yf) θ (xr,yr) = ((xn × c) + (yn × s), (yn × c) - (xn × s))
> where
>    (xn, yn) = (xr - xf, yr - yf)
>    c = cos θ
>    s = sin θ
```

The `transform` function can provide an animal with information about the occupants of its environment, but when dealing with such information calculated for all occupants of the environment, the information is overwhelming. The information is also more precise than one would expect an animal's senses to be. Animals are not aware of the exact relative positions of other objects in the world. Their spatial awareness is less precise than the information a computer can calculate. The `relativeQuadrant` function provides less precise data than the `transform` function about an object's position relative to a focus point, although it makes use of the `transform` function to calculate this data.

The `relativeQuadrant` function tells which quadrant a given point occupies relative to the position and heading of a focus point. Specifically, the function tells which quadrant a point returned by the `transform` function occupies in the transformed relative coordinate plane. So that the function is well defined, `relativeQuadrant` returns a 0 if the positions of the focus point and the relative point are the same. Furthermore, the positive x axis is considered part of the first quadrant, the positive y axis part of the second quadrant, the negative x axis part of the third quadrant and the negative y axis part of the fourth quadrant.

Inputs:

- position of focus point
- orientation/heading of focus point
- position of relative point

```
>relativeQuadrant :: (Floating a, Ord a) ⇒ (a, a) → a → (a, a) → Int
>relativeQuadrant (x_f,y_f) θ (x_r,y_r)
> | (x_f,y_f) ≡ (x_r,y_r) = 0
> | x_n > 0 ∧ y_n ≥ 0 = 1
> | x_n ≤ 0 ∧ y_n > 0 = 2
> | x_n < 0 ∧ y_n ≤ 0 = 3
> | x_n ≥ 0 ∧ y_n < 0 = 4
>   where
>      (x_n,y_n) = transform (x_f,y_f) θ (x_r,y_r)
```

A heading is a Float value, but it represents a position on a unit circle. All points on a circle can be represented by radian values in the range $[0, 2\pi)$. Calculations are simpler when all headings are within this range, therefore confineHeading is used to maintain this invariant. It is a periodic function that maps any Float value to the range $[0, 2\pi)$. It does this by recursively adding or subtracting 2π from the input until the result is inside the desired range.

Inputs:

- radian measure of any size

```
>confineHeading :: (Floating a, Ord a) ⇒ a → a
>confineHeading θ
> | θ < 0 = confineHeading (θ + 2π)
> | θ ≥ 2π = confineHeading (θ - 2π)
> | otherwise = θ
```

2.5.3 Constants

This module contains all of the constant settings that control some aspect of the simulation. The values assigned to the constants are merely potential values. Different sets of results are obtained by setting these constants to different values.

The functions and data structures associated with these constants have not yet been fully explained. This module serves as an introduction to the constants and explains restrictions imposed by the environment, but to fully understand the purpose of each constant, the modules following this one must be read.

```
>module Constants
> (speciesEqualityConstant,
> minMove, maneuverabilityRestriction, moveCostMultiple,
> energyRestriction, deathEnergy, interactionDistance,
> carrionMultiplier, predationMultiplier, predationConstant, cannibalGracePeriod,
> minStartEn, minSight, minManeuverability, minMoveSpeed, minLifespan, minMutationRate,
> minRadius, minMatingCost, minMaturity, minMatingTime, minMatingRecovery,
> maxStartEn, maxSight, maxManeuverability, maxMoveSpeed, maxLifespan, maxMutationRate,
> maxRadius, maxMatingCost, maxMaturity, maxMatingTime, maxMatingRecovery,
> sStartEnergy, sSight, sHerbivore, sCarnivore, sCannibal, sCarrion,
> sManeuverability, sSpeed, sLifespan, sMutation, sRadius, sMateCost,
> sLearning, sMaturity, sMateTime, sDummy, sMateRecovery,
> plantsPerGrowth, growthTimeStep, plantNutrition,
> binGridSize, binGridSizeX, binGridSizeY, binWidth, binHeight,
> initialPopulation, initialPlants, netArchitecture, maxFloatMutation, maxIntMutation,
> initialEnergy,
> seed1, seed2, seed3, seed4, seed5, seed6, seed7, seed8, seed9, seed10,
> metabolicRate, decay) where
```

Interaction Distance

Animals can only interact if they are within a given distance of each other. The exact distance depends on the type of interaction, but the constant controlling this distance is the `interactionDistance`. When one animal wants to eat another, the distance between the two must be less than `r + interactionDistance`, where `r` is the eating animal's radius. When two animals are to mate, the distance must be less than `min(`r_1`,`r_2`) + interactionDistance`, where r_1 and r_2 are the two animals' radii.

The `interactionDistance` cannot be negative, because then it would be impossible for animals to interact.

```
>interactionDistance = 10 :: Float
```

Species Equality Constant

Two animals are considered members of the same species if they have similar trait genes. Two trait genes are considered similar if the difference in their values is less than `speciesEqualityConstant`. If even a single pair of trait genes has a difference of at least `speciesEqualityConstant`, then the two animals are not of the same species.

Because trait genes have a range of [0,1], `speciesEqualityConstant` should be in the range (0,1). This is because any value at most zero would mean that no two animals were of the same species, and any value at least one would mean that every animal would be of the same species.

```
>speciesEqualityConstant = 0.15 :: Float
```

Minimum and Maximum Starting Energy

When an animal is born, one of its two parents gives it its starting energy. This amount is $s \times m$, where s and m are the parent's starting energy and mating cost respectively. The minimum and maximum values allowed for the starting energy trait are `minStartEn` and `maxStartEn`.

The constant `minStartEn` must be positive to prevent an animal from dying the moment it is born, and `maxStartEn` $>$ `minStartEn`. Having `maxStartEn` < 1 assures that some energy is lost in the mating process. A parent cannot perfectly transfer energy to its offspring without some small loss. Furthermore, if the `maxStartEn` were greater than one, then the offspring would gain energy that its parents did not have (the system of energy flow would not be closed).

```
>minStartEn = 0.1 :: Float
>maxStartEn = 0.9 :: Float
```

Minimum and Maximum Sight Range

An animal can only sense an object when its distance from the object is within the animal's sight range. The minimum and maximum values allowed for the sight range trait are `minSight` and `maxSight`.

The constant `minSight` must be positive if an animal is to be able to see anything. The constant `maxSight` must be greater than `minSight`. Because an animal's senses are completely confined to one `Bin` at a time, `maxSight` should not be greater than the length of a `Bin`'s diagonal, because at this length the animal can already sense the entire `Bin` from all positions within the `Bin`. A larger sight range would not increase the range of senses.

```
>minSight = 15 :: Float
>maxSight = 70 :: Float
```

Minimum and Maximum Maneuverability, and Maneuverability Restriction

An animal's maneuverability controls how much it can turn during a time step. The maximum amount that an animal can turn to the left or right on a time step is `m /(r` $\times$ `maneuverabilityRestriction)`, where `m` is the animal's maneuverability and `r` is its radius. This means that an increased radius decreases turning range and an increased maneuverability increases turning range. The minimum and maximum values allowed for the maneuverability trait are the `minManeuverability` and `maxManeuverability`.

The constant `minManeuverability` should be a positive value for the sake of consistency (although, since the difference between a left and right turn is only the sign, negative maneuverability values could be implemented). Zero is not an acceptable value because a maneuverability of zero means an animal can only head straight forward throughout its entire life. The constant `maxManeuverability` must be greater than `minManeuverability` and should be less than π, because for maneuverability values at least equal to π, the ranges for left and right turns begin to overlap, making a larger turn effectively less. The constant `maneuverabilityRestriction` should be positive for consistency, and cannot be zero because that would cause a divide by zero error when calculating the turn range.

```
>minManeuverability = 0.1 :: Float
>maxManeuverability = 3.0 :: Float
>maneuverabilityRestriction = 0.1 :: Float
```

Minimum and Maximum Movement Speed, and Minimum Movement

The maximum distance that an animal can move forward on a time step is its movement speed. The minimum distance that an animal must move on a time step is `minMove`. The minimum and maximum values allowed for the movement speed trait are `minMoveSpeed` and `maxMoveSpeed`.

All movement values must be positive because animals can only move forward and not backwards. The constant `minMove` must be greater than zero to assure that animals move on each time step. The constant `minMoveSpeed` must be greater than `minMove`, and `maxMoveSpeed` must be greater than `minMoveSpeed`.

```
>minMoveSpeed = 1.0 :: Float
>maxMoveSpeed = 5.0 :: Float
>minMove = 0.2 :: Float
```

Movement Cost Multiple

Every movement that an animal makes costs energy. The cost per movement is `d` × `r` × `moveCostMultiple`, where `d` is the distance moved and `r` is the animal's radius.

The constant `moveCostMultiple` must be greater than zero, because a negative value would cause the animal to gain energy with each movement and a value of zero would mean that all movement was free.

```
>moveCostMultiple = 0.05 :: Float
```

Minimum and Maximum Maturity Age

An animal cannot mate until it reaches its maturity age. This means that its age is greater than its maturity age. The minimum and maximum values allowed for the maturity age trait are `minMaturity` and `maxMaturity`.

The maturity age cannot be negative because negative ages have no meaning. The constant `maxMaturity` must be greater than `minMaturity`.

```
>minMaturity :: Integral a ⇒ a
>minMaturity = 30
>maxMaturity :: Integral a ⇒ a
>maxMaturity = 70
```

Minimum and Maximum Lifespan

An animal's lifespan determines how long it can possibly live. Once an animal's age is greater than its lifespan (once it has existed for a number of time steps equal to its lifespan plus one) it dies of old age. The minimum and maximum values allowed for the lifespan trait are `minLifespan` and `maxLifespan`.

The constant `minLifespan` should be greater than `maxMaturity`, so that every animal has a chance to mate. The constant `maxLifespan` must be greater than `minLifespan`.

```
>minLifespan :: Integral a ⇒ a
>minLifespan = 300
>maxLifespan :: Integral a ⇒ a
>maxLifespan = 900
```

Minimum and Maximum Mutation Rate

During mating each of an animal's three chromosomes has the same probability of undergoing mutation (in which one randomly chosen gene of the chromosome is mutated, as explained in the *Genetic Algorithms* section below). This probability is the animal's mutation rate. These three mutation events are independent (or rather, pseudo-independent, given that whether or not they occur depends on values from the same random number generator). The minimum and maximum values allowed for the mutation rate trait are `minMutationRate` and `maxMutationRate`.

Mutation rate is a probability, which automatically restricts all mutation rate values to the range [0,1]. The constant `minMutationRate` must be greater than zero, because if all animals had a mutation rate of zero, then mutation would never occur, and new traits would cease to evolve in the population. The constant `maxMutationRate` must be greater than `minMutationRate`. Because it is a probability, `maxMutationRate` ≤ 1.

```
>minMutationRate = 0.1 :: Float
>maxMutationRate = 1.0 :: Float
```

Minimum and Maximum Radius

Radius describes the size of an animal. Each animal has the shape of a circle. A larger radius increases the range within which an animal can interact with other animals. An animal is only capable of eating another animal with at most the same radius. However, a larger radius decreases turning range and increases movement costs. The minimum and maximum values allowed for the radius trait are `minRadius` and `maxRadius`.

A negative radius would have no meaning, therefore radius cannot be negative. The constant `minRadius` must be greater than zero because a value of zero would cause a divide by zero error when calculating turn range. The constant `maxRadius` must be greater than `minRadius`.

```
>minRadius = 5 :: Float
>maxRadius = 18 :: Float
```

Minimum and Maximum Mating Cost

The amount of energy that an animal expends when it mates is its mating cost. When two animals mate, two offspring are produced. Each parent gives a portion of the energy it expends in mating to one of the offspring. This portion is `s` × `m`, where `s` is the parent's starting energy and `m` is its mating cost. This amount of energy is what the offspring is born with. The minimum and maximum values allowed for the mating cost trait are `minMatingCost` and `maxMatingCost`.

A negative mating cost would make no sense, because then an animal would gain energy from mating. Therefore mating cost must be positive. The constant `minMatingCost` must be greater than zero both so that mating has a cost associated with it and so that the offspring start with some energy. The constant `maxMatingCost` must be greater than `minMatingCost`.

```
>minMatingCost = 50 :: Float
>maxMatingCost = 300 :: Float
```

Minimum and Maximum Mating Recovery Time

The mating recovery time is the number of time steps an animal must wait between matings. When an animal is born, its mating countdown is set to zero. After it mates, this value is set equal to the mating recovery time. The mating countdown decreases by one every time step until it reaches zero again. An animal cannot mate unless its mating countdown equals zero. The minimum and maximum values allowed for the mating recovery time trait are `minMatingRecovery` and `maxMatingRecovery`.

Negative time values have no meaning, so the mating recovery time cannot be negative. The constant `maxMatingRecovery` must be greater than `minMatingRecovery`.

```
>minMatingRecovery :: Integral a => a
>minMatingRecovery = 40
>maxMatingRecovery :: Integral a => a
>maxMatingRecovery = 90
```

Minimum and Maximum Mating Time

Mating time is not actually a time, but a value compared between potential mates to see if they are compatible (this is explained further in the *Animals* section). The mating time can only be one of several discrete values, and two animals can only mate if they have the same mating time value. The minimum and maximum values allowed for the mating time trait are `minMatingTime` and `maxMatingTime`.

The constant `maxMatingTime` must be greater than `minMatingTime`. It is not required, but mating time values are kept positive for simplicity's sake.

```
>minMatingTime :: Integral a => a
>minMatingTime = 0
>maxMatingTime :: Integral a => a
>maxMatingTime = 7
```

Starting Gene Values

The initial population of animals starts out with identical trait genes (except for the sex gene, which is random). All trait values are in the range [0,1], and so are the starting values. The following are the starting gene values in order, sans the sex gene.

```
>sStartEnergy = 0.5 :: Float
>sSight = 0.0 :: Float
>sHerbivore = 0.6 :: Float
>sCarnivore = 0.4 :: Float
>sCannibal = 0.4 :: Float
>sCarrion = 0.4 :: Float
>sManeuverability = 0.0 :: Float
>sSpeed = 0.0 :: Float
>sLifespan = 0.3 :: Float
>sMutation = 1.0 :: Float
>sRadius = 0.0 :: Float
>sMateCost = 0.2 :: Float
>sLearning = 0.1 :: Float
>sMaturity = 0.5 :: Float
>sMateTime = 0.5 :: Float
>sDummy = 0.5 :: Float
>sMateRecovery = 0.5 :: Float
```

Death Energy

An animal dies when its energy level drops below its death point. An animal's death point is `s` $\times$ `deathEnergy`, where `s` is the amount of energy the animal is born with.

The death point of an animal must be less than the energy the animal is born with, so that it gets a chance at life. It must be greater than zero so that a corpse can exist after the animal dies. Therefore `deathEnergy` must by in the range (0,1).

```
>deathEnergy = 0.7 :: Float
```

Energy Restriction

An animal cannot have more energy than its maximum energy. An animal's maximum allowable energy is `s` $\times$ `energyRestriction`, where `s` is the amount of energy the animal is born with.

The maximum energy must be greater than the energy the animal is born with so that the animal is able to gain energy after being born. Therefore `energyRestriction` must be greater than one.

```
>energyRestriction = 21 :: Float
```

Predation Multiplier and Constant

When one living animal eats another, the animal being eaten loses energy which is transferred to the animal doing the eating. This amount is (`predationMultiplier` $\times$ $|r_1 - r_2|$) + `predationConstant`, where r_1 and r_2 are the radii of the two animals. The difference in the radii of the two animals can be any value at least zero. If the difference is zero (meaning that the predator and prey are the same size), then `predationConstant` assures that the predator still gets some energy from eating. The bigger the predator is, the more energy it can get from eating. However, the prey only gives energy up to a max of its total energy level, and because its energy is then zero, it is removed from the environment on the next time step, as though it was swallowed whole.

The constant `predationMultiplier` must be greater than zero to assure that a predator receives energy when it eats. The constant `predationConstant` must be greater than zero to assure that a predator receives some energy even when eating an animal of the same size.

```
>predationMultiplier = 11 :: Float
>predationConstant = 100 :: Float
```

Carrion Multiplier

When a living animal eats a dead one, energy is transferred from the corpse to the carrion eater. The amount of energy transferred is `r` $\times$ `carrionMultiplier`, where `r` is the radius of the carrion eater. The corpse only loses up to a max of all the energy it has.

The constant `carrionMultiplier` must be greater than zero to assure that carrion eaters receive energy when they eat carrion.

```
>carrionMultiplier = 15 :: Float
```

Cannibal Grace Period

When an animal is born there is a given number of time steps during which no cannibal of the same species will eat it. This is the constant `cannibalGracePeriod`. This discourages cannibals from eating their own offspring. At the very least they cannot do it right away.

Times cannot be negative, so `cannibalGracePeriod` is non-negative.

```
>cannibalGracePeriod :: Integral a ⇒ a
>cannibalGracePeriod = 40
```

Plant Growth Time Step

Besides the regular time steps, during which animals act, there are also plant growth time steps. The constant `growthTimeStep` controls how often they occur. The simulation maintains a counter that is incremented on every normal time step. The `loop` function in the *Simulation* section has a case for normal time steps (the animals act) and for plant growth time steps. Whenever the loop counter is evenly divisible by `growthTimeStep`, the simulation allows a new batch of plants to grow. Otherwise a normal time step for animals is carried out. The check of divisibility is actually a check to see if `n` mod `growthTimeStep` ≡ `0`, where `n` is the loop counter. Because the counter does not increment on a plant growth time step, a Boolean value is also checked to see if one occurs. After the plant growth time step this value is set to `False`, to indicate that no plant growth time step needs to be performed, even though the counter is still divisible by `growthTimeStep`. The next regular time step resets this value to `True`, so that plants will grow the next time a plant growth time step comes along.

The constant `growthTimeStep` cannot be zero because it causes a divide by zero error when modulus division is used. Negative values would work, but are unnecessary. The constant `growthTimeStep` should be positive.

```
>growthTimeStep :: Integral a ⇒ a
>growthTimeStep = 5
```

Potential New Plants Per Growth Time Step

On every plant growth time step a given number of random positions in a `PlantGrid` (explained in *Plants*) are generated, and the same number of determining factors are also generated. The number of values generated is `plantsPerGrowth`. For each random position generated, the area around that position is examined to calculate the probability of a plant growing there. Each of these probabilities is compared against a determining factor to see if a plant grows at that position. This is repeated for each `Bin`. Therefore the maximum number of plants that could potentially grow within one `Bin` on a single growth time step is `plantsPerGrowth`.

The constant `plantsPerGrowth` must be greater than zero so that plants have the chance to grow.

```
>plantsPerGrowth :: Integral a ⇒ a
>plantsPerGrowth = 100
```

Plant Nutrition

The amount of energy that an animal earns from eating a plant is `plantNutrition`. When an animal eats a plant its own energy increases and the plant disappears.

The constant `plantNutrition` must be positive so that an animal gains energy from eating plants.

```
>plantNutrition = 65 :: Float
```

Bin Grid Size

The `BinGrid` described in the *Bins* section is a `BRAMatrix` that holds `Bin`'s, which together make up the environment of the simulation. The tuple `binGridSize` is the size of the grid. The number of columns in the grid is `binGridSizeX` and the number of rows in the grid is `binGridSizeY`.

Both `binGridSizeX` and `binGridSizeY` must be greater than zero because there can be no negative number of rows and columns, and if either were zero there would be no `Bin`'s in which to run the simulation.

```
>binGridSize :: (Int,Int)
>binGridSize = (binGridSizeX,binGridSizeY)
>binGridSizeX = 3 :: Int
>binGridSizeY = 3 :: Int
```

Bin Width and Height

Every Bin in the BinGrid is of the same size. The width of every Bin is the binWidth and the height of every Bin is the binHeight.

Because binWidth and binHeight are measurements, they must both be positive. Furthermore, each must be evenly divisible by 10, so that the Bin's can be evenly divided into 10 by 10 squares to accommodate a PlantGrid.

```
>binWidth :: Num a ⇒ a
>binWidth = 100
>binHeight :: Num a ⇒ a
>binHeight = 100
```

Initial Population

The number of animals created in order to start the simulation is initialPopulation. These animals are uniformly distributed among the Bin's in the BinGrid. If initialPopulation is not evenly divisible by the number of Bin's then the extra animals are discarded.

The constant initialPopulation must be positive so that there are animals to observe. It should be evenly divisible by the number of Bin's in the BinGrid, because for a given number of Bin's n, only initialPopulation div n animals are used anyway.

```
>initialPopulation = 150 :: Int
```

Initial Plants

At the start of the simulation an empty world is made. Animals are added to it. Then the simulation goes through a given number of plant growth steps before the simulation starts, in order to give the world an initial population of plants. The number of plant growth time steps executed before the start of the simulation is initialPlants.

Negative numbers are not allowed, because there is no such thing as a negative number of plant growth steps.

```
>initialPlants = 200 :: Int
```

Neural Network Architecture

In general, the architecture of a neural network describes both the number of nodes per layer and the manner in which those nodes are connected. All neural networks defined for this simulation are assumed to be fully connected, so all that is needed to describe the network architecture is the number of nodes per layer. The number of initial inputs is also needed. The neural network architecture is a list of the number of nodes per layer in a neural network. The first value is the number of initial inputs, which is followed by the number of nodes in the input layer, then the number of nodes in each subsequent hidden layer, and finally the number of nodes in the output layer, which is also the number of final outputs.

Animals receive as input information about 6 types of objects from 4 quadrants, making for a total of 24 external inputs. Animals are also aware of their own age and energy, which adds 2 more inputs. All together an animal can sense 26 inputs, so the first value in the neural network architecture must be 26. The outputs of the neural network are used to determine how much an animal turns and how much it moves forward, making for a total of 2 outputs. Therefore the last value in the neural network architecture list must be 2. All values in between the first and last must be positive integers. There is no bound on the number of layers.

```
>netArchitecture = [26,26,26,2] :: [Int]
```

Maximum Float Type Mutation

When mutation occurs, a randomly generated value is added to the preexisting value of a random gene. Both the traits and synaptic weights chromosomes are composed of Float values. If mutating a trait gene causes it to be greater than one, then it is set to one. If mutation causes it to be less than zero, then it is set to zero. The synaptic weights chromosome does not restrict the values of its genes. The randomly generated value that mutates a gene can be anywhere in the range [`-maxFloatMutation, maxFloatMutation`].

The constant `maxFloatMutation` cannot be zero, because then mutation would never occur (or when it did occur, nothing would change). Mutation values are kept positive for consistency.

```
>maxFloatMutation = 0.3 ::  Float
```

Maximum Int Type Mutation

Mutation is the same for the time delays chromosome, except that it contains `Int` values and is therefore mutated by `Int` type mutations. A value that mutates a gene in a time delays chromosome can be an integer in the range [`-maxIntMutation, maxIntMutation`].

The constant `maxIntMutation` must be non-zero to allow mutation to occur, and should be positive for the sake of consistency.

```
>maxIntMutation = 3 ::  Int
```

Initial Energy

Animals that are born receive energy from their parents at birth, but the first animals to populate the simulation's environment have no parents and are artificially placed into the world. Instead of receiving energy from their parents, the amount of energy they start with at birth is `initialEnergy`.

The constant `initialEnergy` must be positive so that the starting animals have a chance to live.

```
>initialEnergy = 200 ::  Float
```

Metabolic Rate

It costs energy to make energy. An animal loses energy every time step based on its size and on how complex its diet is (how many diet traits it has). An animal can have up to four diet traits: herbivore, carnivore, cannibal and carrion eater. The amount of energy an animal loses per time step is `metabolicRate` $\times$ r^n, where `r` is the animal's radius and `n` is the number of diet traits the animal has.

The constant `metabolicRate` must be greater than zero to assure that the animal loses energy from its metabolism. If it were zero there would be no cost, and if it were negative the animal would actually gain energy for no reason.

```
>metabolicRate = 0.05 ::  Float
```

Decay Amount

On every time step a dead animal loses a given amount of energy as a result of rotting away. The amount that it loses is `decay`. This continues every time step until the dead animal's energy level is at most zero, and it is removed from the environment.

The constant `decay` must be positive because if it were negative the corpses would gain energy until it was time to come back to life. The value of `decay` cannot be zero because then the corpses would never go away, unless eaten by carrion eaters.

```
>decay = 0.3 ::  Float
```

Random Number Generator Seeds

Several random number streams are used in the simulation. Each one requires its own seed. The random number streams are designed such that the period of values is equal to the number of possible values. This means that the stream cycles through every possible value in its range before repeating. It also means that any given value in this range is equally valid as a starting seed. Thus the seeds below are all equally valid in that the values they produce are as random as one can hope for from a linear congruential pseudo-random number generator.

All that is required of each seed is that it be at most the maximum value of the random number generator and at least zero. The maximum value of the random number generator is 65535 (which is `modulus - 1`, as described in *Streams of Random Numbers*).

The seeds below are used to generate random values for the following purposes:

1. Generates x and y coordinates used to determine the starting positions of the initial population of animals.
2. Generates random start headings in radians for the animals of the initial population.
3. Generates the starting synaptic weights for the initial population of animals.
4. Makes a stream of determining factors for various events.
5. Makes a stream of random positions in the traits chromosome.
6. Makes a stream of random positions in the weights and time delays chromosomes (they both have the same length).
7. A stream of `Float` type mutations.
8. A stream of `Int` type mutations.
9. Random x coordinates for plants to grow at.
10. Random y coordinates for plants to grow at.

```
>seed1 = 17235 ::  Integer
>seed2 = 27341 ::  Integer
>seed3 = 34197 ::  Integer
>seed4 = 57122 ::  Integer
>seed5 = 16429 ::  Int
>seed6 = 23417 ::  Int
>seed7 = 37712 ::  Integer
>seed8 = 44214 ::  Int
>seed9 = 56713 ::  Int
>seed10 = 13721 ::  Int
```

2.6 Organisms

2.6.1 Overview

The simulation contains two types of organisms: plants and animals. Only one type of plant exists, and it is defined in the *Plants* module. As for animals, only one type is defined, but it can take many forms. The population of animals can evolve into multiple populations, each with different eating habits and behaviors. This evolution can occur thanks to the use of a genetic algorithm defined in *Genetic Algorithms*, which comes after *Plants*. The animals themselves are defined in *Animals*, which is the third and final module of this section. It should be noted in advance that the code in *Animals* is particularly complex.

2.6.2 Plants

The focus of the simulation is on the artificial animals and their evolutionary development as a population, but in order to support this population there is a basic source of sustenance at the bottom of the food chain. This food source is called a plant. Plants are always at the bottom of the food chain in nature because, except for a few fascinating exceptions, they do not feed upon other organisms to survive. Instead, plants are nourished by water and minerals from the soil and rays from the sun. Assuming that these resources are available, plants will continue to grow, but even plants must compete with each other for resources.

The more plants there are growing in a given patch of land, the more resources that are needed to sustain them. Plants that are better adapted to absorbing nutrients from the soil, and that maximize their potential to absorb sunlight, are more likely to survive and produce offspring. A plant's ability to absorb nutrients depends on the structure and properties of its roots, and plants often maximize sun absorption by growing large leaves and by growing as tall as possible, so as not to be blocked from the sun by other plants.

There are several options available when it comes to simulating such plants. Height and nutrient absorbency could be traits that affect the lifespan of a plant. These traits could be compared between nearby plants to see how the limited supply of resources is divided among the competing plants. These traits could also be derived from genes that get passed down from generation to generation, thus allowing the plants to evolve. Another option for representing the competition for resources between plants is to make their environment with cellular automata, such that when clusters of plants get too large they start to die out because of the lack of resources.

While it would be interesting to implement and study such phenomena, the primary focus of the simulation is on the artificial animals of the population rather than the artificial plants. The features considered above would add additional complications to the simulation, and the resulting costs in both space and time complexity would be extremely detrimental to the study of the artificial animals in the environment. Therefore a much simpler scheme is applied for the growth of plants in the environment.

The environment only supports a limited number of plants, and the only distinguishing feature between plants is position. No two plants can share the same position. Every plant is represented by a square 10 by 10 units in size that occupies a single cell within a `BRAMatrix`. Each `BRAMatrix` is contained within an instance of the `Bin` data structure, which is explained fully in the section *Bins*. A collection of `Bin`'s spans the whole environment. The number of plants is limited because each `BRAMatrix` only stores a finite number of elements. Each cell within a `BRAMatrix` corresponds to a 10 by 10 square in a `Bin`, such that the cells are disjoint and partition the `Bin` (this means that both the height and width of the `Bin` must be divisible by 10).

The growth model for the plants is simplified as follows: The environment starts with randomly scattered plants. Every time that `growthTimeStep` number of time steps pass (whenever the number of time steps is divisible by `growthTimeStep`) there is a chance for up to `plantsPerGrowth` number of plants to come into existence. For each potential plant a random position within the given `BRAMatrix` is chosen. If that cell already contains a plant, then no new plant grows there. Otherwise the probability of a plant growing there is $0.1 + (n \times 0.1)$, where n is the number of plants in the neighboring cells. Put another way, the base probability of a plant growing in a cell is 0.1 and increases by 0.1 for each plant in a neighboring cell up to a max of 0.9, because every cell has eight neighbors. The neighbors of cells on the edges of a `BRAMatrix` are considered to be those cells directly opposite on the other edge. This means that plants are more likely to grow near each other in clusters. Determining factors are used to decide whether a plant grows in the given cell or not.

The following module models plant growth as described above and also provides a way for the animals of the environment to eat the plants.

```
>module Plants
> (module MyBRAMatrix,
> PlantGrid, newPlantGrid,
> countPlants,
> executeGrow,
> coordToArray, arrayToCoord, visiblePlants) where
```

Plants are aspects of the environment, and therefore make use of code from the *Environment* section. Plant growth is probabilistic, adding an element of chance and uncertainty. Several constants are used to control plant growth, which occurs within the confines of `BRAMatrix`'s.

```
>import Environment -- Environment
>import Probability -- Probability
>import Constants -- Constants
>import MyBRAMatrix -- Binary Random Access Matrix
```

The only distinguishing feature between plants is position. Otherwise any two plants are identical. Furthermore, a plant can only occupy one of a discrete number of positions within a `BRAMatrix`. This means that every plant can be represented by a simple Boolean variable stored at the appropriate index within a `BRAMatrix`. This way information about position is maintained by the `BRAMatrix`, and need not be represented by a separate plant object. The resulting representation is so simple as to be defined as a type alias instead of a data structure. The `PlantGrid` type is an alias for a Boolean `BRAMatrix`.

```
>type PlantGrid = BRAMatrix Bool
```

The `newPlantGrid` function makes use of the `BRAMatrix` command `initialize2D` to create a new `PlantGrid` containing no plants. That means that every cell within the `PlantGrid` has a value of `False`. The size of the `PlantGrid` is determined by a 2-tuple given as input.

Inputs:

- 2-tuple for the width and height of the resulting grid

```
>newPlantGrid :: (Int,Int) → PlantGrid
>newPlantGrid size = initialize2D size False
```

For the sake of data collection there is a function that counts the number of plants in a given `PlantGrid`. This is a linear operation that counts the number of occurrences of `True` within a `PlantGrid`. The positions of the plants being counted are irrelevant, so the `elems2D` function is used to recast the `PlantGrid` elements into a simple list, thus stripping all information about position. This list is easier to manipulate recursively than a `BRAMatrix`. The helper function `addUp` does the actual work of counting the plants.

Inputs:

- the `PlantGrid` whose plants are to be counted

```
>countPlants :: PlantGrid → Int
>countPlants g = addUp (elems2D g)
>    where
>       addUp :: [Bool] → Int
>       addUp [] = 0
>       addUp (b:bs)
>          | b = 1 + (addUp bs)
>          | otherwise = addUp bs
```

The chances of a given grid cell sprouting a plant depend on the number of plants in neighboring cells. Therefore the `countNeighbors` function is needed to count these neighbors. A single index within a `PlantGrid` is taken as input, and the eight cells surrounding the given cell are referenced by adding or subtracting 1 from the x and y coordinates in every combination possible (as shown in the table). The `confineArrayIndex` function is used to prevent errors from arising when referencing border cells. This results in cells on the opposite borders being counted as neighbors. This is not fully accurate, because the actual neighbors of a given border cell are in a different `PlantGrid` (the exception being an environment containing only one `PlantGrid`). This is another small sacrifice made for the sake of representational and computational simplicity.

$(x-1, y+1)$	$(x, y+1)$	$(x+1, y+1)$
$(x-1, y)$	(x, y)	$(x+1, y)$
$(x-1, y-1)$	$(x, y-1)$	$(x+1, y-1)$

Coordinates of neighboring matrix cells relative to the center cell with position (x,y).

Two simple helper functions are used in counting the neighboring plants. The `btn` function converts `True` and `False` to 1 and 0 respectively. The function is used by the `pln` function, that returns a 1 if a given cell contains a plant and a 0 otherwise. The main function adds up a series of `pln` calls to each of the cells neighboring the cell of interest, giving a result from 0 to 8.

Inputs:

- a `PlantGrid`
- a position within the `PlantGrid`, whose neighbors are to be counted

```
>countNeighbors :: PlantGrid → (Int,Int) → Int
>countNeighbors g (x,y) =
> (pln (x - 1, y - 1)) + (pln (x, y - 1)) + (pln (x + 1, y - 1)) +
> (pln (x - 1, y)) + (pln (x + 1, y)) +
> (pln (x - 1, y + 1)) + (pln (x, y + 1)) + (pln (x + 1, y + 1))
>    where
>       btn :: Bool → Int
>       btn True = 1
>       btn False = 0

>       pln :: (Int,Int) → Int
>       pln (x,y) =
>          btn (lookup2D (confineArrayIndex (matrixSize g) (x,y)) g)
```

Each plant in a neighboring cell increases a plant's growth chances by 0.1, unless the cell in question is already occupied. The `getGrowthChance` function makes a probability distribution, as described earlier in the *Probability* section, representing this situation. After the cell is checked to assure it does not yet contain plant, the `countNeighbors` function is used to make an appropriate distribution. By default, a value of 1 for the probability 0.1 is assigned to the event of a plant growing, and the value 9 for the probability 0.9 is assigned to the no growth event. The number of neighboring plants is added to 1 and subtracted from 9 to obtain the adjusted chances of growth and no growth respectively. These values are used to make the corresponding distribution.

Inputs:

- a `PlantGrid`
- the position of a cell within the `PlantGrid` for which to determine the growth chances

```
>getGrowthChance :: PlantGrid → (Int,Int) → [ProbEvent Bool]
>getGrowthChance g (x,y)
> | lookup2D (x,y) g = makeDistribution [False] [1]
> | otherwise = makeDistribution [True,False] [1 + adj, 9 - adj]
>    where
>       adj = (fromInteger.toInteger) (countNeighbors g (x,y))
```

The `executeGrow` function makes use of all the functions above to have plants grow. It takes a `PlantGrid` as input, modifies it, and returns the modified `PlantGrid` as output. The modifications made depend on two lists of data that are sent to the function as input. One is a list of `Int` 2-tuples corresponding to positions in the `PlantGrid`, and the second is a list of determining factors. Both lists should contain randomly generated values. Furthermore, both lists must be of equal length, since each value in one list corresponds to a value in the other. The function uses the BRAMatrix's massUpdate2D command using a list of update values generated by the helper function gPos.

The gPos function goes through the two lists recursively and generates a list of update tuples: positions in the grid coupled with new values for those positions. For each value in the list of position tuples, the getGrowthChance function generates the probability distribution for the event of a plant growing there. Then the next value from the list of determining factors is used with the pickEvent function from the *Probability* section to determine whether or not a plant grows. If the result is True, then an entry for the current cell position is added to the list of updates to be made, and execution continues with a recursive call to gPos. If the result is False, then the next recursive call executes without any updates being made. This continues until both the list of cell positions and the list of determining factors are empty.

Inputs:

- a PlantGrid
- a list of randomly generated positions within the PlantGrid
- a list of determining factors for plant growth

```
>executeGrow ::  PlantGrid → [(Int,Int)] → [Float] → PlantGrid
>executeGrow g ps cs = massUpdate2D g (gPos ps cs)
>    where
>       gPos ::  [(Int,Int)] → [Float] → [((Int,Int),Bool)]
>       gPos [] [] = []
>       gPos ((x,y):ps) (c:cs)
>          | val = ((x,y),True):(gPos ps cs)
>          | otherwise = gPos ps cs
>             where
>                val = pickEvent (getGrowthChance g (x,y)) c
```

The second half of this module deals with how plants are eaten. To be eaten is the purpose of plants in the environment. This is another reason why a simplistic plant growth model is justified. The plants are constantly eaten by the animals, to the point that adding more restrictions and complications to the growth of plants would be excessive. Plants can only be removed from the environment by being eaten. Otherwise they are immortal. Of course, a plant would only be able to avoid being eaten indefinitely if no animals existed, at which point it would be pointless to run the simulation any longer. The animals do a well enough job of keeping the plant population in check without the necessity of additional impediments.

The first issue to be resolved when plants and animals interact is the issue of different coordinate systems. The artificial animals move in continuous (within the limits of a digital computer) two-dimensional space, while the plants have discrete positions within two-dimensional matrices. As mentioned previously, every cell in a PlantGrid corresponds to a 10 by 10 square in the continuous environment occupied by the artificial animals. What was not mentioned earlier is that most of these squares are created by crossing two half-open intervals with each other. This means, for example, that the square in cell (0,0) contains all points whose x-coordinate is at least 0, but strictly less than 10 (hence the interval is closed at 0 but open at 10). The y-coordinate is similarly restricted. This is necessary, because otherwise the borders of the cells would either overlap or be undefined. What is needed is a surjective ("onto") mapping from the continuous space to the square cells of the matrix. Building the squares from half-open intervals solves the problem at the cell borders, but creates a new problem at the edges of the PlantGrid (or rather, at two of them). Now the PlantGrid is seemingly open at its upper and rightmost edges, but this is not the case. Instead, cells on the upper border are closed at both their upper and lower edges, and cells on the right border are closed at both their left and right borders. This is done to prevent an exception from occurring when an animal wanders precisely onto the upper or rightmost edge of the PlantGrid. This representation gives rise to the function coordToArray.

The coordToArray function converts from points in continuous two-dimensional space to points in discrete matrix space. The function uses a case structure, of which the first three cases deal with exceptions at the boundaries of the PlantGrid. The general case uses the idea of squares made from half-open intervals explained above. The `floor` of each coordinate divided by 10 is taken. This works fine unless a coordinate is on the upper or rightmost barrier, in which case the resulting index would be out of bounds. These special cases are caught by comparing the x and y coordinates to the boundaries of the `PlantGrid`. They are fixed by subtracting 1 from the offending coordinate(s).

Inputs:

- a tuple for a point in continuous two-dimensional space

```
>coordToArray :: (Float,Float) → (Int,Int)
>coordToArray (x,y)
>    | top ∧ side = ((floor (x/10)) - 1, (floor (y/10)) - 1)
>    | top = (floor (x/10), (floor (y/10)) - 1)
>    | side = ((floor (x/10)) - 1, floor (y/10))
>    | otherwise = (floor (x/10), floor (y/10))
>      where
>         top = y ≡ binHeight
>         side = x ≡ binWidth
```

The `arrayToCoord` function performs the opposite conversion, going from discrete space to continuous space. Some accuracy is lost in this conversion. Every cell occupies an area of continuous space as opposed to a single point, so the point that is returned by the function is chosen to be the center of the square. This is obtained by multiplying each matrix index by 10 and then adding 5 to it. The point obtained by multiplying each of the coordinates by 10 is the lower left corner of the cell area. Adding an additional 5 to each coordinate shifts the point to the center of the cell, because 5 is half the length and width of the 10 by 10 cells.

Inputs:

- a tuple for a `PlantGrid` index

```
>arrayToCoord :: (Int,Int) → (Float,Float)
>arrayToCoord (x,y) = con ((x × 10) + 5, (y × 10) + 5)
> where
>    con :: (Int,Int) → (Float,Float)
>    con (a,b) = ((fromInteger.toInteger) a, (fromInteger.toInteger) b)
```

In order to seek out plants, animals need knowledge of where they are. All plants have an absolute position in the environment, but of more importance to an animal is the position of plants relative to the animal. Every animal has an absolute position as well as a heading, which is the direction the animal is facing. Because the ease with which an animal can move to a certain location is affected by the direction it is currently facing, it is pertinent for the animal to consider the locations of plants relative to its heading as well as its position. The `visiblePlants` function below provides this information.

The function takes a `PlantGrid` as input and returns information about plant locations relative to a focus point in the environment. This point has an associated heading and a limited range of sight. The information gained about plant locations is general rather than exact. Only plants within the range of sight of the focus point can be seen. This means that the distance between the focus point and a plant's center must be less than the range of sight. This reduces the area of interest to a disc around the focus point with a radius equal to the range of sight. This disc is further divided into four quadrants of interest. The quadrants arise from the intersection two lines: the line of sight of the point and this line's perpendicular through the focus point.

The information returned about each quadrant is the number of plants it contains. These four numbers are returned within a list. Although this information is limited, it is enough to provide the animals with an idea of where the highest, as well as the lowest, concentration of plants is within the immediate environment.

The work of the function is performed by its helper function `collect`. The `collect` function receives the list of position-value tuples for the `PlantGrid` and an empty accumulator for storing the number of plants found in each quadrant. In order to take full advantage of Haskell's lazy execution, the two special cases are checked first, thus reducing computation. Every cell in the `PlantGrid` must be checked, so the first test is to see if the cell contains a plant. This is accomplished by checking a single Boolean value. In the absence of a plant at the given cell, execution continues recursively. Should a plant exist at the given cell, its distance from the focus point is then calculated. If the distance is greater than the range of sight, then the plant is disregarded and execution continues recursively. Otherwise the accumulator needs to be updated before execution continues.

The accumulator is updated by the helper function `toQ`. First the position of the plant relative to the focus point's heading and position is calculated by the function `relativeQuadrant`. This value, along with the accumulator, is sent to the function `toQ`, and case analysis determines which of the four values in the accumulator needs to be incremented.

Once all cells have been checked, the accumulator is returned as a list of four elements. All in all the function runs in linear time, meaning that it takes quadratic time for each animal in the population to call it.

Inputs:

- a `PlantGrid`
- a focus point in the continuous space encompassed by the `PlantGrid`
- the orientation of the focus point
- the focus point's range of sight

```
>visiblePlants :: PlantGrid → (Float,Float) → Float → Float → [Float]
>visiblePlants g (x,y) h r = [q1, q2, q3, q4]
> where
>   (q1, q2, q3, q4) = collect (assocs2D g) (0.0,0.0,0.0,0.0)

>   collect :: [((Int,Int),Bool)] → (Float,Float,Float,Float) → (Float,Float,Float,Float)
>   collect [] (q1, q2, q3, q4) = (q1, q2, q3, q4)
>   collect (((x1, y1),a):as) (q1, q2, q3, q4)
>      | not a = collect as (q1, q2, q3, q4)
>      | distance (x,y) (xt,yt) > r = collect as (q1, q2, q3, q4)
>      | otherwise = collect as (toQ (q1, q2, q3, q4) q)
>        where
>           q = relativeQuadrant (x,y) h (xt, yt)
>           (xt, yt) = arrayToCoord (x1, y1)

>           toQ :: (Float,Float,Float,Float) → Int → (Float,Float,Float,Float)
>           toQ (q1, q2, q3, q4) 1 = (q1 + 1, q2, q3, q4)
>           toQ (q1, q2, q3, q4) 2 = (q1, q2 + 1, q3, q4)
>           toQ (q1, q2, q3, q4) 3 = (q1, q2, q3 + 1, q4)
>           toQ (q1, q2, q3, q4) 4 = (q1, q2, q3, q4 + 1)
>           toQ (q1, q2, q3, q4) _ = (q1, q2, q3 , q4)
```

2.6.3 Genetic Algorithms

Genetic algorithms are based on the idea of evolution through natural selection, which is largely accredited to Charles Darwin. His theory is based on the following ideas: organisms of a population are similar but different; these differences can be passed from parents to their offspring; resources are scarce and organisms are in competition for these scarce resources; organisms with traits conducive to survival and mating are more likely to survive and have offspring. Therefore, traits which improve chances of surviving and mating are more likely to be passed down to offspring, and are therefore more likely to persist in a population throughout time. Since all organisms are in competition and resources are scarce there is always pressure from the environment to change and improve. When populations of organisms change as a whole it is called evolution. The scarcity of resources and pressure of competition are elements of natural selection.

In Darwin's time the mechanism of inheritance was not yet fully understood. Scientists knew that offspring shared certain traits with their parents, but had no way of predicting which traits would be inherited and which would not. The one to answer these questions was Gregor Mendel, who ironically was a contemporary of Darwin, but did not make his work widely known. He discovered the basic rules of heredity and inheritance which eventually became the foundation of modern genetics.

It is now known that all living organisms contain DNA (Deoxyribonucleic acid), and that within DNA reside genes. Genes are organic molecules located at specific locations in DNA. The position of a gene is called its locus. Each gene at each locus is responsible in some way for determining the traits of an organism. A chromosome is a long, continuous strand of DNA. An organism can have several chromosomes, each containing a particular set of genes associated with particular traits of the organism.

Sexually reproducing organisms have two copies of each chromosome within each of their cells: one from the mother and one from the father. Such cells are termed diploid. In order to reproduce, such organisms first create haploid cells through the process of meiosis. Haploid cells contain only one chromosome of each type. The genes of these chromosomes are a mixture of genes from both parents, because during meiosis the chromosomes go through the process of crossover. Each pair of like chromosomes is brought together and then separated such that the resulting chromosomes contain genetic material from one parent up to a certain locus and genetic material from the other parent from then on. Sets of like chromosomes are then separated from each other into haploid cells, called gametes. Each gamete contains only one chromosome of each type. During mating, gametes are exchanged and fuse to form a zygote. A zygote is a diploid cell composed of one set of chromosomes from each parent, making for two of each type. The zygote eventually develops into a fully grown organism, using the DNA instructions from the chromosomes of both parents as a basis for growth.

Traditional genetic algorithms use an abstraction of this process in order to find solutions to complex problems. To use a genetic algorithm, one must first be able to formulate a proposed solution to the problem as a string, or several strings, of data. The type of data depends on the problem domain, but it could be anything from binary data to ASCII characters to floating point numbers. Such a string is called a chromosome. A given set of chromosomes contains all of the data needed to describe a proposed solution to a problem. A proposed solution, which is a set of chromosomes, is an individual organism. There also needs to be a method or heuristic for measuring how close a proposed individual is to an actual solution. Such a method is called a fitness function.

To start the genetic algorithm, a preferably large number of individuals are randomly generated. The fitness function is applied to every member of the population, and then some members of the population are allowed to mate and reproduce, with preference given to individuals with better fitness scores. Most algorithms also remove a certain number of organisms with bad fitness scores from the population. Exactly how these individuals are chosen depends on the algorithm. There are many different methods of selection, each with its own advantages and disadvantages. The actual process of mating is an abstracted version of meiosis. The organisms do not have two of each type of chromosome, so the process of crossover is slightly different. Instead of performing crossover on like chromosomes of a single organism, genetic algorithms generally take like chromosomes from two mating organisms and perform crossover on them instead, using one or both of the results to create new offspring, all of which have only one chromosome of each type. The crossover point is randomly chosen [8].

Another element common to both real world mating and genetic algorithms is the chance for mutation. Mutation allows for the introduction of new genes, and therefore new traits, into a population. These traits can be either helpful or detrimental, but if they are helpful then it is expected that natural selection will favor them. In genetic algorithms there are several ways to model mutation. One must consider the rate of mutation, how many times mutation can occur per mating, and how an existing gene value is updated as a result of mutation. The gene value can either be replaced by a completely new value or modified by a given function. The modification method is used in this module. Gene values are numbers, and the modifying function is addition.

However, this project does not make use of a typical genetic algorithm. Instead of using a concrete fitness function, the simulation stays in line with the ideas of evolution by natural selection and assumes that traits conducive to survival and reproduction will naturally be selected for. The simulation also does not make use of global population updates in the same manner that typical genetic algorithms do. Instead, animals mate and die as suits their individual circumstances, much like in the real world. However, the genetic algorithm used in this simulation does maintain the practice of having only one chromosome of each type per individual. The artificial animals are therefore haploid, although they sexually reproduce.

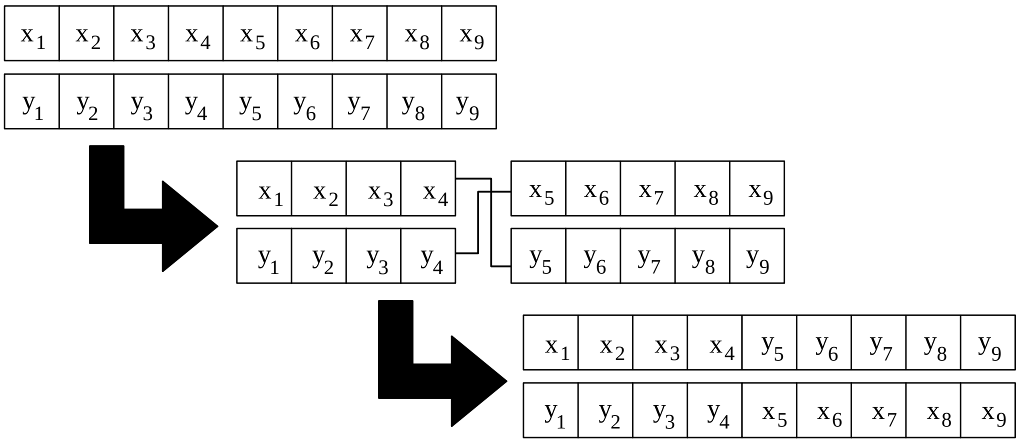
Functions that deal with the chromosome manipulation aspect of genetic algorithms are developed below. The mating of animals is handled in the *Animals* section.

```
>module Genetic
> (Chromosome,
> crossover, mutate,
> mateGenes) where
```

A chromosome is a list of data of a given type. Though it is possible to make a `Chromosome` instance of any type, only numeric types are used in this simulation. This is required by the functions below, so that addition can be used as the modifying function during mutation.

```
>type Chromosome a = [a]
```

The `crossover` function takes two chromosomes, performs genetic crossover on them at a specified point and then returns both results. It works by swapping elements in the two chromosomes while decrementing the crossover position. When it reaches 0 the two chromosomes are returned with everything after to crossover point left unmodified. For this function to work properly it is important that both chromosomes be of the same type and length and that the crossover point not exceed that length. The crossover point is intended to be random.



x1 x2 x3 x4 x5 x6 x7 x8 x9
y1 y2 y3 y4 y5 y6 y7 y8 y9
x1 x2 x3 x4
y1 y2 y3 y4
x5 x6 x7 x8 x9
y5 y6 y7 y8 y9
x1 x2 x3 x4 y5 y6 y7 y8 y9
y1 y2 y3 y4 x5 x6 x7 x8 x9

When crossover is performed on two lists, the resulting lists receive a portion of their genes from each list.

Inputs:

- tuple of like chromosomes from two parents
- crossover position (within bounds and random)

```
>crossover ::  (Chromosome a,Chromosome a) → Int → (Chromosome a,Chromosome a)
>crossover (c1, c2) 0 = (c1, c2)
>crossover (m:ms, f:fs) n = (f:c1, m:c2)
> where
>    (c1, c2) = crossover (ms, fs) (n − 1)
```

The method of mutation used in the simulation is very dynamic. The mutation rate is a trait of each animal, and therefore is potentially different for each application of the `mutate` function. During mating, each chromosome has a chance of having one of its genes mutated. Mutation occurs if the rate of mutation is at least the value of a determining factor. If mutation does occur, then it does so at a locus specified in the input. This value is meant to be a random number within the bounds of the chromosome. As mentioned above, mutation occurs by modification. One of the inputs is a random mutation value, which is directly added to the existing gene value if mutation occurs. This assures that traits change gradually instead of suddenly.

Inputs:

- chromosome with numeric contents
- rate of mutation
- random gene locus
- determining factor for mutation
- modification to gene if mutation occurs

```
>mutate ::  (Num a, Floating b, Ord b) ⇒ Chromosome a → b → Int → b → a → Chromosome a
>mutate c rate pos ran new
> | rate < ran = c
> | otherwise = mut c pos new
> where
>    mut ::  Num a ⇒ Chromosome a → Int → a → Chromosome a
>    mut (c:cs) 0 δ = ((c + δ):cs)
>    mut (c:cs) n δ = c:(mut cs (n - 1) δ)
```

Mating is composed of both the actions of crossover and mutation. The `mateGenes` function combines both functions into one. First crossover is performed between two parent chromosomes, then the `mutate` function is called with each resulting chromosome. Notice that each call to `mutate` has a different set of inputs, allowing mutation to occur differently in each of the resulting offspring.

Inputs:

- tuple of like numeric chromosomes from two parents
- two rates of mutation
- two random gene loci
- two determining factors for mutation
- crossover position
- two gene modifications for when mutation occurs

```
>mateGenes ::(Num a, Floating b, Ord b) ⇒ (Chromosome a,Chromosome a) → (b,b) →
>              (b,b) → (Int,Int) → Int → (a,a) → (Chromosome a,Chromosome a)
>mateGenes (m,f) (p1,p2) (c1,c2) (m1,m2) cross (n1,n2)
>    = (mutate nm p1 m1 c1 n1, mutate nf p2 m2 c2 n2)
>       where
>          (nm, nf) = crossover (m,f) cross
```

2.6.4 Animals

The animals defined in this module are the most important aspect of the simulation. Their behavior and interactions are what is being studied. Since they are so important, the code dealing with them is relatively complex and intricate. The module first defines several data structures which compose the `Animal` data structure, and then goes on to define a myriad of operations on animals and groups of animals.

```
>module Animal
> (speciesEq, deriveSpecies,
> numberOfTraits,
> Traits, traitGenes, sex, sight, diet, maneuverability,
> maxSpeed, lifespan, mutationRate, radius, mateCost,
> learningRate, matingTime, matingRecovery,
> take01toBool, take01toFloat, take01toInt, formDiet,
> Body, weightGenes, delayGenes, hebbTimeNet,
> Animal, idN, position, heading, age, energy, traits,
> body, lastAction, matingCountdown, maxEnergy, deathPoint,
> niceShowTraits, niceShowBody, niceShowAnimal,
> livingAnimals, deadAnimals,
> newHebbTimePopulation,
> masterHebbTimeAction) where
```

The simulation makes use of Hebbian time delayed neural networks, which control the behavior of the animals. All outside stimulus is filtered through an animal's neural network to determine how it acts. The *Genetic* module is needed to support the crossover and mutation operations involved in mating. The *Environment* module defines the world in which the animals live. The *Plants* module allows the animals to interact with plants, which serve as a basic food source. Code from *Constants* and *Standard Code Extensions* provides a few additional bits and pieces needed in several functions.

```
>import HebbTimeNet -- Neural Networks Supporting Discrete Time Delays and Hebbian Learning
>import Genetic -- Genetic Algorithms
>import Environment -- Environment
>import Plants -- Plants
>import Constants -- Constants
>import StandardExts -- Standard Code Extensions
```

The `Animal` data structure is composed of several smaller data structures, which are complex in their own right. Before defining the `Animal` data type, several smaller component data structures will be defined. The first of these is the `Traits` data structure.

Traits

In common speech a trait can be nearly any aspect of an animal. However, we define a trait to be an entity within an instance of the `Traits` data structure. Only prominent numerically definable traits are defined in this structure. Traits related to behavior, or that arise as a combination of other traits are not explicitly defined. Also notable is that the `Traits` data structure is not directly related to the composition of an animal's neural network (the weights and time delays of the network). The effect of certain weights or time delays on the functionality of a neural network, and thus an animal's behavior, is very hard to determine. The traits defined within the `Traits` data structure are transparent, in that one can see clearly how a modification to one of these traits will affect an animal. In this work, the word trait will refer to one of the specific traits defined in the `Traits` data structure as opposed to more nebulous traits that are not so easily defined and measured.

The following are the key defining traits of an animal:

Starting Energy: With the exception of the initial animals in the population, each animal is born from two parents mating. Both parents expend energy when they mate, and a portion of this energy is that with which the offspring starts life. The starting energy trait defines what portion of this expended energy becomes the offspring's. It is strictly less than 100%. This assures that some energy is lost in mating.

Sex: As with animal species that can be divided into male and female, the animals of this simulation can have one of two possible sexes. What these sexes are, male and female, is irrelevant, and so sex is represented by a Boolean value. The two types of sexes correspond to `True` and `False`. Sex is important in mating, because an animal can only mate with a member of the opposite sex. However, the sex trait plays no role outside of mating.

Sight: This is the range within which the animal can sense other objects in the environment. Sensable objects are animals and plants. Animals use their sight to seek out food (plants and other animals), mates and also to avoid predators. The more an animal can see, the more it can sense, although a large visual range could potentially overload an animal's senses.

Diet: Diet is actually four separate traits that are conceptually related. Each is a Boolean value telling whether or not the animal exhibits a certain eating behavior. The four eating types are herbivore, carnivore, cannibal and carrion eater. Herbivores can eat plants. Carnivores can eat animals of other species (species defined below) and cannibals can eat animals of the same species. In either case, animals can only eat animals that are not bigger than they are. Carrion eaters can eat the dead bodies of animals that remain rotting in the environment. Although these traits are conceptually related, they are otherwise completely independent of one another. It is possible for an animal to develop any combination of the four eating behaviors described above, but having the flexibility of multiple eating behaviors has its downside. Being able to digest more types of food requires more energy. This energy cost raises exponentially with the number of eating behaviors an animal has.

Maneuverability: Each animal has a chance to turn and move on every time step. The maximum number of radians that it can turn in either direction on one time step is determined by its maneuverability in combination with the animal's size (radius, see below).

Maximum Movement Speed: Going along with maneuverability is maximum speed. It is something of a misnomer however, since it is actually a distance measurement. It can be thought of as a speed in that it is the maximum distance that an animal is capable of covering per one time step. Since the time steps are discrete rather than continuous, the animal actually jumps from point to point without traversing the space in between. However, these spaces are so small such that the illusion of smooth movement in maintained.

Lifespan: Normally an animal continues to live as long as its energy is greater than its death point (see below). When the energy level drops to or below the death point, the animal is dead, but an animal can also die if it lives for a long period of time. This death comes completely independent of the animal's energy level. The lifespan trait is the maximum number of time steps that an animal can live. Once an animal has survived this many time steps it dies regardless of its energy level.

Mutation Rate: This is a percentage chance that a random mutation will occur during the mating process. This value can become very small, but it never shrinks to zero. There must always be a chance, no matter how small, of mutation.

Radius: This is basically the size of the animal. For simplicity's sake, all animals have the shape of circles within the two-dimensional environment. Therefore radius perfectly describes the space an animal occupies. Size is mainly important in determining whether one animal can eat another. Bigger animals have the advantage, but being big increases the rate at which an animal consumes energy and limits the animal's ability to turn, so large size has its downside as well.

Mating Cost: The amount of energy the animal must expend in order to mate. A portion of this energy is what is given to the animal's offspring to start life with.

Maturity Age: This trait describes the age, number of time steps, an animal must reach before it is able to mate. Before an animal is allowed to mate and pass on its genes it must first pass the test of time and survive to reach the value set by its maturity age. A high maturity age could be a problem for a population, since it is likely that fewer animals could survive to that age. However, a low maturity age would allow relatively weaker animals to pass on their genes, thus weakening the gene pool over time.

Learning Rate: The learning rate trait is used by the animal's neural network in the Hebbian learning process. In this process it serves as the learning rate parameter, as explained earlier in the section *Neural Networks Supporting Discrete Time Delays and Hebbian Learning*.

Mating Time: Sometimes animals of a certain species will only mate under certain special conditions such as during a given time of day or year, or in a certain climate. Sometimes animals will only mate if certain mating rituals are carried out. These preferences can make it so that otherwise compatible animals never mate, or never even have the chance to mate with each other. These traits play a role in the selection process, but they otherwise do not affect the animal. The mating time trait is an *abstraction* of such traits. It has no effect on the animal's behavior, but it limits the partners that the animal can choose from. Mating time can take on one of a discrete number of values. In order for two animals to mate they must have the same mating time value. The idea is that the two animals tend to mate at the same time of day or during the same season as each other (hence the name), but this trait imposes no actual restriction on when the animal can mate. Time is only in the name as a reference to the idea behind the gene's creation.

Mating Recovery: This trait actually does impose a time restriction on when an animal can mate. Specifically, this trait is the number of time steps an animal must wait after successfully mating before it can mate again.

Dummy Trait: Animals also have a trait that has no real purpose. It is simply a dummy value that gets passed on to offspring like any other trait. In spite of this trait's seeming uselessness, it actually does play a role in mating. Like all traits (except for sex), this dummy trait plays a role in determining an animal's species. Two animals can only mate if they are of the same species, and the dummy trait means that two otherwise similar animals may be of different species. This is yet another mechanism that can have unexpected effects on the natural selection process.

All together that makes 18 traits.

```
>numberOfTraits ::  Int
>numberOfTraits = 18
```

All data within the `Traits` data structure is derived from a list of `Float` values between 0 and 1 inclusive, called the traits chromosome. The traits chromosome is stored within the `Traits` data structure under the name `traitGenes`.

All of the above mentioned traits are represented within the traits chromosome, but not all are present within the `Traits` data structure. Only those values that need to be frequently accessed are data members of the `Traits` data structure. As data members, the traits are stored in their proper units as opposed to the uniform `Float` representation that the genes of the traits chromosome have.

The two traits not represented within the `Traits` data structure are starting energy and the dummy trait. The dummy trait is not present for obvious reasons: it is not used for determining any aspect of the animal's behavior. The starting energy is not present because it is only needed when the animal is first born. Afterwards the energy level fluctuates according to the environment and how the animal reacts to it, regardless of what the starting energy was. All other traits are present.

```
>data Traits
> = Learn {traitGenes::Chromosome Float, sex::Bool, sight::Float,
>    diet::(Bool,Bool,Bool,Bool), maneuverability::Float, maxSpeed::Float, lifespan::Int,
>    mutationRate::Float, radius::Float, mateCost::Float, maturity::Int,
>    learningRate::Float, matingTime::Int, matingRecovery::Int}
>       deriving (Show, Eq)
```

A default `Show` instance is derived for troubleshooting purposes, but for a clearer view of the data, `niceShowTraits` can be used. It only displays the traits chromosome. Since everything else can be derived from that, it is sufficient.

Inputs:

- a `Traits` instance

```
>niceShowTraits ::  Traits → String
>niceShowTraits x = show (traitGenes x)
```

The values in the traits chromosome are all `Float` values in the range [0,1], but the data members in the `Traits` data structure have more variety. The following functions convert from a trait's gene representation in the traits chromosome to a more meaningful representation.

The `take01toBool` function converts trait genes to Boolean values. Values less than 0.5, and therefore in the range [0,0.5), map to `False`. Values of at least 0.5, and therefore in the range [0.5,1], map to `True`. This function is used for the sex gene and all four diet genes.

Inputs:

- a gene from the traits chromosome (`Float` value in the range [0,1])

```
>take01toBool ::  Float → Bool
>take01toBool x
>    | x < 0.5 = False
>    | otherwise = True
```

The `take01toFloat` function expands the range of a `Float` value to something other than [0,1]. The input `lo` becomes the new minimum value and the input `hi` becomes the new maximum value. Naturally, `lo` should be less than `hi`. The sight, maneuverability, maximum speed, mutation rate, radius and mating cost traits make use of this function. Although the learning rate trait is also a `Float` value, it does not use this function since its natural range is [0,1].

Inputs:

- a gene from the traits chromosome (`Float` value in the range [0,1])
- the new minimum value after conversion
- the new maximum value after conversion

```
>take01toFloat ::  Float → Float → Float → Float
>take01toFloat x lo hi = (x × (hi - lo)) + lo
```

The `take01toInt` function both adjusts the range of the input and changes the type from `Float` to `Int`. It makes use of the `take01toFloat` function. The `take01toFloat` function converts the range, and then the `round` command assures that the end result is of type `Int`. Since the inputs to the function are of the `Int` type, but the `take01toFloat` function takes `Float` values as input, the `convert` function (see *Standard Code Extensions*) is used to convert these values. The lifespan, maturity, mating time and mating recovery traits are recast in this way.

Inputs:

- a gene from the traits chromosome (`Float` value in the range [0,1])
- the new minimum value after conversion
- the new maximum value after conversion

```
>take01toInt ::  Float → Int → Int → Int
>take01toInt x lo hi = (fromInteger.round) (take01toFloat x (convert lo) (convert hi))
```

It has been stressed that trait genes are within the range [0,1]. Some operations, particularly mutation, can violate this invariant. The `restrictTrait` function below maintains the invariant by chopping off any value greater than 1 at 1, and any value less than 0 at 0.

Inputs:

- a gene from the traits chromosome that could possibly be out of range

```
>restrictTrait ::  Float → Float
>restrictTrait x
>    | x > 1 = 1
>    | x < 0 = 0
>    | otherwise = x
```

The `formDiet` function creates the `diet` data member by returning a tuple of results from `take01toBool`.

Inputs:

- a tuple of the four diet genes in the traits chromosome

```
>formDiet ::  (Float,Float,Float,Float) → (Bool,Bool,Bool,Bool)
>formDiet (a,b,c,d) = (herb,carn,cann,carr)
> where
>    herb = take01toBool a
>    carn = take01toBool b
>    cann = take01toBool c
>    carr = take01toBool d
```

The next function is the one responsible for making an instance of `Traits` out of a traits chromosome. The traits chromosome is a list, but a list of a specific length. The `buildTraits` function must receive as input a trait chromosome of the correct length, which is 18. The order of the genes in the traits chromosome is as follows:

1. Starting Energy
2. Sex
3. Sight
4. Diet - Herbivore
5. Diet - Carnivore
6. Diet - Cannibal
7. Diet - Carrion Eater
8. Maneuverability
9. Maximum Movement Speed
10. Lifespan
11. Mutation Rate
12. Radius
13. Mating Cost
14. Learning Rate

15. Maturity Age
16. Mating Time
17. Dummy Trait
18. Mating Recovery

Knowing this, the `buildTraits` function performs the proper conversion function on each gene, and then assigns the result to its proper place in the `Learn` constructor for the `Traits` data structure.

Inputs:

- a traits chromosome (a length 18 list of `Float` values in the range [0,1])

```
>buildTraits ::  Chromosome Float → Traits
>buildTraits (se:a:b:c:d:e:f:g:h:i:j:k:l:m:n:o:p:q:[])
> = Learn (se:a:b:c:d:e:f:g:h:i:j:k:l:m:n:o:p:q:[]) sx si di mn ms ls mu fs mc ma m mt mr
>    where
>       sx = take01toBool a
>       si = take01toFloat b minSight maxSight
>       di = formDiet (c,d,e,f)
>       mn = take01toFloat g minManeuverability maxManeuverability
>       ms = take01toFloat h minMoveSpeed maxMoveSpeed
>       ls = take01toInt i minLifespan maxLifespan
>       mu = take01toFloat j minMutationRate maxMutationRate
>       fs = take01toFloat k minRadius maxRadius
>       mc = take01toFloat l minMatingCost maxMatingCost
>       ma = take01toInt n minMaturity maxMaturity
>       mt = take01toInt o minMatingTime maxMatingTime
>       mr = take01toInt q minMatingRecovery maxMatingRecovery

>buildTraits _ = error ‘‘Trait chromosomes must have 18 genes’’
```

Species

The species of an animal is derived from its `Traits` instance. The concept of species is important because only animals of the same species can mate with each other. However, the definition of *same species* needs to be clarified. The animals in the population do not start with a predefined species. In fact, there are no predefined species at all. The concept of species is completely dynamic. It is a relative relationship between two animals.

At first this seems illogical, but it is really the only way that new species can emerge over time. The first member of a new species is sure to die out unless it can still mate with members of the species that spawned it. However, if this new animal really represents a new species, then it must be different from some of the members of the species that spawned it. It can only mate with members of this species that share some of its distinguishing traits. As more offspring are born which are similar to the new species, but dissimilar to the original one, a new species emerges that can no longer interbreed with the first species. Biologists call this sympatric speciation.

Another type of speciation, for which there are more and better documented precedents in the real world, is allopatric speciation. Allopatric speciation is based on the idea that populations of the same species that somehow become geographically isolated from each other will with time develop into different species incapable of interbreeding with each other. A long time apart allows the two populations to evolve in different directions to the point that they would no longer interbreed, should the two populations be rejoined. However, if the time apart is too short, then the differences may not be large enough, and the two populations could possibly once again become a single species.

This type of speciation is not possible in this simulation, because although the environment is divided into `Bin`'s, no portion of the environment is inaccessible to any of the animals. There is no way that two populations could become separated.

This begs the question of how sympatric speciation actually occurs. If a population is confined within the same geographic area, then why would not all large genetic differences fade over time? How can multiple species arise? The answer is that new species should be able to arise if behaviors or traits evolve that restrict who an animal will mate with. In this way, even populations that live in close contact would not be able to mate, and would therefore be able to evolve into different species.

The mating time and dummy trait genes were added specifically for this purpose, but all of the genes have a role in limiting the mating options of the population. The species of an animal is considered to be represented by its traits chromosome sans the sex trait gene. Since in order to mate, a species requires members of both sexes, it would not make any sense if sex had an influence on an animal's species. The `deriveSpecies` function retrieves the species information from the `Traits` instance of an animal.

Inputs:

- a `Traits` instance

```
>deriveSpecies :: Traits → [Float]
>deriveSpecies t = dSpe (traitGenes t)
> where
>    dSpe (se:sx:ts) = (se:ts)
>    dSpe _ = error ''Not enough traits to derive species.''
```

As stated before, the concept of species is relative between individuals. Two animals are considered to be of the same species, and thus able to mate, if all of their genes are sufficiently close to one another. "Sufficiently close" is defined by the value of `speciesEqualityConstant`. For two animals, each gene of the species representation is compared to the corresponding gene of the other animal. If the difference between the two genes is less than `speciesEqualityConstant` for each pair of genes, then the two animals are considered to be of the same species. This form of equality is not transitive. This means that if one animal is the same species as two other animals, it does not necessarily mean that these two animals are of the same species. Animals that are between two species, like in the example above, are what allow new species to come into being. Once all of these border animals have died out, the two populations can no longer exchange genes, and thus there are two separate species. Of course, it is possible that this could never happen. Whether or not it does is one of the questions of interest for this simulation.

In any case, it is the `speciesEq` function that performs the comparison between two animals. Given the `Traits` of two animals, it uses `deriveSpecies` on both to get the pertinent genes, and then compares the results with a recursive series of checks strung together by Boolean "and" predicates. If the difference between any pair of genes is at least `speciesEqualityConstant`, then the two animals are incompatible and `False` is returned. Otherwise they are of the same species and `True` is returned.

Inputs:

- the `Traits` instances of two animals

```
>speciesEq :: Traits → Traits → Bool
>speciesEq a b = sEq (deriveSpecies a) (deriveSpecies b)
> where
>    sEq :: [Float] → [Float] → Bool
>    sEq [] [] = True
>    sEq (t:ts) (s:ss) = ( | t - s | < speciesEqualityConstant) ∧ (sEq ts ss)
```

Body

The next major component of an animal is its body, which contains the parts of the animal controlling action. The primary role of a `Body` instance is to house an animal's neural network. The `Body` instance also stores the two chromosomes that help define the neural network, these being the chromosomes for synaptic weights and for discrete time delays. The weight genes are stored in `weightGenes` and the time delays are stored in `delayGenes`. The neural network is in `hebbTimeNet`, which is of course a Hebbian time delayed neural network. The only constructor is `HebbTime`.

```
>data Body
> = HebbTime {weightGenes::Chromosome Float, delayGenes::Chromosome Int, hebbTimeNet::Net}
>    deriving Show
```

Bodies have a default `Show` instance for troubleshooting purposes and the `niceShowBody` function for data collection purposes. This function shows the synaptic weights and the time delays of the neural network.

Inputs:

- a `Body` instance

```
>niceShowBody ::  Body → String
>niceShowBody x
> = ‘‘Weights:  ’’++show (weightGenes x)++‘‘\n’’++‘‘Delays:  ’’++show (delayGenes x)
```

A `Body` is composed of two chromosomes and a neural network. The neural network is derived from the two chromosomes, but an additional piece of data is also required. Namely, the architecture of the neural network. The architecture is defined as the number of nodes per layer and the way they are connected. The networks in this simulation are assumed to be fully connected (every node on one layer connects to every node on the next), so the only piece of data needed is the number of nodes per layer. This is provided in the form of a list of `Int` type, where each value is the number of nodes for that layer (the number of inputs is considered a layer).

To create a `Body`, the `buildHebbTimeBody` function is used. It takes as input the two chromosomes and a list describing the neural network's architecture. The `netFromLists` function is then used to create the neural network.

Inputs:

- a chromosome of synaptic weights
- a chromosome of time delays
- a list defining the network architecture

```
>buildHebbTimeBody ::  Chromosome Float → Chromosome Int → [Int] → Body
>buildHebbTimeBody ws ds is = HebbTime ws ds (netFromLists is ds ws)
```

The chromosome containing discrete time delays stores values of type `Int`. This allows them to take on any valid `Int` value, but a proper time delay should be at least zero. Anything less than zero would cause an error, so much like with the genes in the traits chromosome, a function is needed to maintain the invariant. The `restrictTimeDelay` function does this by setting time delays to zero whenever they drop down to a negative number.

Inputs:

- a discrete time delay that may be out of bounds

```
>restrictTimeDelay ::  Int → Int
>restrictTimeDelay x
>    | x > 0 = x
>    | otherwise = 0
```

Animal

Having defined both the `Body` and `Traits` of an animal, the next step is to define the `Animal` data type itself. The two constructors for animals are `Live` and `Dead`. All animals start out being created with the `Live` constructor, and when the animal dies the `Dead` constructor is used to create what is technically a new animal, but it represents the corpse of the animal that died. Corpses stay in the environment until they rot away, or become completely eaten by carrion eaters.

One data member common to both constructors is `idN`, which stands for ID number. Every animal is born with a unique identifying number. When an animal dies, its corpse retains the ID number. The dead animal has the same ID number as its living counterpart.

Actually, every data member of the `Dead` constructor is shared by the `Live` constructor, though sometimes the meaning is slightly different. Both constructors have a member for position, which is an xy-coordinate. The position given is a position within the `Bin` that the animal occupies. For living animals this value changes every time step, because living animals are required to move. Once an animal dies, this value never changes. The corpse stays where it died until it disappears all together.

Both constructors also have `energy` as a data member. For a living animal, energy is a representation of the animal's health. As long as the animal's energy is above its death point it continues to live. The death point is a data member of living animals only, which is derived from the energy the animal receives from its parents at birth. This value multiplied by `deathEnergy` is the animal's death point (`deathEnergy` is between zero and one). Maintaining energy is a constant battle for animals. Every movement that the animal makes causes it to lose energy proportional to the distance traversed and its size. In order to replenish energy the animal must eat of the several food sources available, which provide varying amounts of energy depending on various factors. The goal (in terms of natural selection) is for the animal to survive long enough and gather enough energy so that it can eventually mate. This process also costs the animal a large quantity of energy. Should the animal continually gain more energy than it loses, it will eventually find that it can gain no more energy. This threshold is also defined by the energy the animal receives at birth, and is a data member of `Live` animals called `maxEnergy`. The maximum energy is the product of the birth energy and `energyRestriction`.

Dead animals have energy as well, though it is at most the animal's death point. A corpse rots a little bit on every time step. This means that it loses a little energy every time step, until its energy reaches zero. Carrion eaters can take away some of this energy by eating the corpse. When a dead animal's energy is at or below zero it has completely rotted away, and no longer exists in the environment.

The last data member of the `Dead` constructor (and therefore the last that it has in common with the `Live` constructor) is `lastAction`. This is an `Int` code that describes the action performed by the animal on the previous time step. This data member plays no role in the simulation and is purely for data collection purposes. By monitoring this data member, one can see what types of behaviors an animal is exhibiting, and also tell how it died. For obvious reasons, dead animals do not do much, but there are still codes which pertain specifically to dead animals as well as codes that pertain specifically to living animals.

A dead animal is capable of having one of the following values for its `lastAction` data member:

-5 : Passed Away A dead animal with this code has died of old age, meaning that its age was equal to its lifespan.

-4 : Starved This animal has run out of energy. More accurately, its energy level has dropped to or below its death point.

-3 : Rotted This animal has already been dead for at least one time step, and now does nothing but rot on every time step.

-2 : Eaten A dead animal with this code has died of being eaten by another animal.

The possible `lastAction` values for a living animal are as follows:

-1 : Born This animal has been born, and has yet to perform any actions.

0 : Nothing Besides turning and moving as is required of all living animals, this animal did nothing during the last time step.

1 : Mated This animal mated with another animal to produce offspring during the last time step.

2 : Ate Plant The animal is a herbivore, and ate a plant on the last time step.

3 : Ate Animal of Different Species The animal is a carnivore, and ate a member of another species on the last time step.

4 : Cannibalized Another Animal This cannibal ate a member of its own species on the last time step.

5 : Ate Carrion This carrion eater ate of a corpse on the last time step.

6 : Failed At Mating This animal attempted to mate on the last time step, but for some reason no offspring was produced.

The remaining data members are possessed by living animals only. The heading of an animal tells which way it is facing. It is a value in radians from 0 to 2π. Heading is important in determining how an animal senses its surroundings. It is also the direction in which the animal is currently moving. Age is another data member that only living animals have. It starts at zero when the animal is born, and increases by one on every regular time step (but not on plant growth steps). When this value is equal to the animal's lifespan, the animal dies.

Living animals also have a mating countdown. It starts out at zero when the animal is born. As long as this value equals zero the animal is capable of mating (assuming all other preconditions are met). When the animal mates with another animal, this value is set equal to the animal's mating recovery trait. At this point the mating countdown decreases by one on every time step, and the animal must wait for it to equal zero before it can mate again.

The final two data members of a living animal are those so thoroughly defined above: an instance of `Traits` and an instance of `Body`.

```
>data Animal
> = Live {idN::Integer, position::(Float,Float), heading::Float, age::Int, energy::Float,
>    traits::Traits, body::Body, lastAction::Int, matingCountdown::Int,
>    maxEnergy::Float, deathPoint::Float}
> | Dead {idN::Integer, position::(Float,Float), energy::Float, lastAction::Int}
>    deriving Show
```

The `niceShowAnimal` function displays the pertinent information of an `Animal` instance in a format cleaner than that produced by the default `Show` instance. There is a case for `Dead` type animals and `Live` type animals. The `Dead` case starts with a "D" and then shows the contents of each data member with spaces in between. The `Live` case is similar, except it starts with an "L", excludes the `Body` from the output and displays the contents of its `Traits` with `niceShowTraits` instead of the default `show` function.

Inputs:

- an `Animal`

```
>niceShowAnimal ::  Animal → String
>niceShowAnimal (Dead i p e la) =
>    ‘‘D ’’++(show i)++‘‘ ’’++(show p)++‘‘ ’’++(show e)++‘‘ ’’++(show la)++‘‘\n’’
>niceShowAnimal (Live i p h a e t _ la mc me dp) =
>    ‘‘L ’’++(show i)++‘‘ ’’++(show p)++‘‘ ’’++(show e)++‘‘ ’’++(show la)++‘‘ ’’++(show a)
>    ++‘‘ ’’++(show h)++‘‘ ’’++(show mc)++‘‘ ’’++(show me)++‘‘ ’’++(show dp)++‘‘ ’’
>    ++(niceShowTraits t)++‘‘\n’’
```

A simple but useful instance of `Eq` is defined for animals. Two animals are the same if they have the same ID number. Since ID numbers are unique to each animal, this means that each animals is equal to itself and no other.

```
>instance Eq Animal where
>    a ≡ b = idN a ≡ idN b
```

It is often useful to group animals in terms of the Live and Dead constructors. This is because living animals and dead animals have very different roles, and some functions only apply to animals of one type or the other. The livingAnimals function returns all animals of the Live type within a list of animals and the deadAnimals function returns all animals of the Dead type within a list of animals.

Inputs:

- list of Animal's

```
>livingAnimals ::  [Animal] → [Animal]
>livingAnimals [] = []
>livingAnimals ((Dead {}):xs) = livingAnimals xs
>livingAnimals (x:xs) = x:(livingAnimals xs)
```

Inputs:

- list of Animal's

```
>deadAnimals ::  [Animal] → [Animal]
>deadAnimals [] = []
>deadAnimals ((Live {}):xs) = deadAnimals xs
>deadAnimals (x:xs) = x:(deadAnimals xs)
```

In order to create a new animal, be it at the start of the simulation or after mating, the newHebbTimeAnimal function is used. The animal is of the Live type, the constructor for which is very complicated in comparison with this function. There is no function for Dead type animals because the Dead constructor is relatively simple.

The first input to newHebbTimeAnimal is the quantity of energy it inherits from its parents. For the animals placed in the environment at the start of the simulation this value is instead initialEnergy. The animal's energy data field is set to this value, and it is also used (as described above) to determine the death point and maximum energy of the animal.

The next input is the animal's ID number, which must be unique. The Integer type is used because its potential size is unbounded, which is important when the simulation has run for a long time. In order for the numbers to stay unique, they must become very large.

The next two inputs are position and heading. These values are assigned to the data members of the same names. At the start of the simulation, these values are assigned randomly to the starting animals. Otherwise both position and heading are derived from the parents. Offspring are born in close proximity to their parents.

The next three inputs are the defining inputs of the animal: the traits, synaptic weights and time delays chromosomes. The traits chromosome is used by buildTraits to create the Traits for the animal, and the other two chromosomes are used along with the final input, the neural network architecture, in a call to buildHebbTimeBody to provide the animal's Body.

Other values that are set by the constructor, but which remain the same regardless of input, are age, mating countdown and last action. Both age and mating countdown are set to zero. The last action is set to negative one, the code for being born.

Inputs:

- energy provided by parents
- unique ID number
- xy-coordinate position
- heading in radians

- traits chromosome
- synaptic weights chromosome
- time delays chromosome
- list describing the network architecture

```
>newHebbTimeAnimal ::  Float → Integer → (Float,Float) → Float
>    → Chromosome Float → Chromosome Float → Chromosome Int
>    → [Int] → Animal
>newHebbTimeAnimal se i p h ts ws ds is =
> Live i p h 0 se (buildTraits ts) bod (-1) 0 (energyRestriction×se) (deathEnergy×se)
>    where bod = buildHebbTimeBody ws ds is
```

The simulation starts with a population of animals scattered randomly throughout the environment. This population is homogenous, meaning that every member is of the same species. In fact, each member of the starting population has nearly identical traits. All trait genes are the same except for the sex trait gene. The make up of the synaptic weights and time delays chromosomes are completely random, but every animal shares the same neural network architecture. Every animal is given a unique ID number, placed in a random position, and given a random heading.

One of the purposes of the simulation is to see if diversity can arise from a homogenous population. At the start of the simulation, all animals are very similar to each other, and the entire population is capable of interbreeding. They are all of the same species. If sympatric speciation occurs, then two if not more distinct populations should emerge which are incapable of interbreeding. Hopefully, each of these separate populations will be successful in its own right. These separate populations will no doubt be in conflict with each other, but if they could establish an equilibrium, then it would make for a wonderful example of how evolution and natural selection keep nature in balance.

The function that makes the initial population of homogenous animals is `newHebbTimePopulation`. Its first input is `n`, the number of animals to make. The next few inputs are lists derived from streams of values, the first one being a list of ID numbers for the animals. The other lists hold randomly generated values. So that there is an ID number for each animal made, the ID number list has a length of `n`. The next list holds coordinate positions, but it holds both x and y coordinates, so every animal needs two values from this list. Its length is 2`n`. The next list contains random start headings for the animals, and has a length of `n`. The next two lists are considerably bigger. The first holds random neural network weights and the second holds neural network time delays. The starting synaptic weights are in the range [-1,1]. This means that taking the absolute value of one of these values returns a number in the range [0,1], which is the range of trait genes. Because of this, values are also taken from this list to determine the sex of the animals. This saves the need of having an additional random number stream. Therefore the length of this list is `n` × (1 + `numSs`), where `numSs` is the number of synapses in the neural network. This value is derived by `countSynapses`. The length of the list of time delays is `numSs` × `n`. The final input is the list describing the neural network architecture.

The work of the function is performed by the helper function `nPop`, which produces one animal per recursive call to itself until `n` animals have been made. The animals are made with the `newHebbTimeAnimal` function above. Each animal is given `initialEnergy` as its birth energy, and each animal has a nearly identical traits chromosome called `ts`. On every iteration this chromosome has a different sex gene, which is the absolute value of the first element in the list of synaptic weights. The other genes in the chromosome are always the same and come from the section *Constants*. Only `numSs` number of genes are needed from both the weights chromosome and time delays chromosome per neural network, so `splitAt` is applied to the list, and what remains gets sent to the recursive call with the rest of the extra data.

Inputs:

- number of `Animal`'s to make
- list of unique ID numbers
- list of random x and y coordinates

- list of random headings
- list of random synaptic weights
- list of random time delays
- the list describing the neural network's architecture

```
>newHebbTimePopulation ::  Int → [Integer] → [Float]
>    → [Float] → [Float] → [Int] → [Int] → [Animal]
>newHebbTimePopulation n is ps hs $g_F s$ $g_I s$ ns = nPop n is ps hs $g_F s$ $g_I s$
> where
>  numSs = countSynapses ns

>  nPop ::  Int → [Integer] → [Float] → [Float] → [Float] → [Int] → [Animal]
>  nPop 0 _ _ _ _ _ = []
>  nPop n (i:is) (x:y:ps) (h:hs) (sx:$g_F s$) $g_I s$ =
>   (newHebbTimeAnimal initialEnergy i (x,y) h ts ws ds ns):(nPop (n - 1) is ps hs $r_W s$ $r_D s$)
>     where
>        ts =
>           [sStartEnergy, abs sx, sSight, sHerbivore, sCarnivore, sCannibal, sCarrion,
>           sManeuverability, sSpeed, sLifespan, sMutation, sRadius, sMateCost, sLearning,
>           sMaturity, sMateTime, sDummy, sMateRecovery]
>        (ws,$r_W s$) = splitAt numSs $g_F s$
>        (ds,$r_D s$) = splitAt numSs $g_I s$
```

Reproduction

From animals being artificially produced, the next step is to describe how animals reproduce naturally. How does sexual reproduction occur in the simulation? There are several restrictions on mating. In order to mate, two animals must be of the same species but opposite sex. They must both be mature, their ages are greater than their maturity age traits, and they cannot have mated recently (their mating countdowns must be zero). Both animals must have energy levels greater than their respective mating costs, and they must share the same value for their mating time traits. Finally, if all the above requirements are met, the two animals still need to be in close proximity to each other before they can mate. This long list of requirements assures that mating is not easy. Natural selection assures that those animals that end up mating have found some way to overcome these difficulties.

The animals in this simulation are quite unusual, in that they are in some ways modeled on animals but do not have the genetic structures of animals. Real animals receive half of their chromosomes from their mother and the other half from their father. Every chromosome from the mother's set corresponds to a chromosome in the father's set. The offspring has two versions of each type of chromosome. The ways in which these chromosomes interact is very complex, but many of the basic principles were discovered long ago by Gregor Mendel.

In many ways Mendel is the father of modern genetics, and his work has since been developed upon. One of the basic methods of inheritance that he discovered deals with dominant and recessive traits. Animals have two versions of each gene. Some simple traits are associated with a single gene found at a certain position in a certain chromosome. A recessive trait is one that only exhibits itself when an animal has two copies of the gene it is associated with; that means that the animal got a copy of the recessive gene from both parents. There need only be one copy of a dominant gene in an animal's genetic code for a dominant trait to exhibit itself. There are also many traits that have varying degrees of expression rather than being simply either/or. Sometimes many different genes contribute to the expression of a certain trait, and sometimes an exhibited trait is a combination of either/or and varying representation relationships.

This is all very interesting, but such means of genetic expression are not possible for this simulation's animals as they are defined in this module. This is because each instance of `Animal` only stores a single copy of each chromosome. In this respect the animals are actually somewhat more like fungi than animals, but in terms of the behaviors these animals are capable of, they are like animals. Another reason that the chromosome representation of this model does not allow for recessive and dominant traits is that the trait genes take on values within a continuous range rather than a discrete range, as real genes do.

In spite of the fact that these animals do not have a typical genetic structure, they still utilize a form of reproduction in which crossover of genes occurs. Whenever two animals mate, two offspring are produced. Each parent has one copy of each type of chromosome: traits, weights and time delays. When the parents mate, each chromosome of the one parent undergoes crossover with the corresponding chromosome of the other parent. The crossover of two chromosomes creates two new chromosomes, each of which goes to one of the two new offspring. In this way the benefits of crossover are maintained without the necessity of having twice as many chromosomes per animal.

After crossover occurs, each of the newly created chromosomes has a chance of being mutated. The mutation rate is a trait gene of the traits chromosome and can be modified through mutation. Whenever a gene mutates, the new gene is a slight modification of the previous gene, rather than a completely new one. This helps assure that the evolution process is a gradual one.

Offspring enter the world as fully formed, fully grown animals at conception. They must reach their maturity age before they can mate, but other than that they are no different (no weaker, slower or smaller) than the other animals in the environment. This is yet another difference from real world animals, which is allowed for the sake of simplicity. Even without gestation and child phases, the animals still have plenty of opportunities to interact and compete with each other.

Before there can be offspring, mating must occur. The myriad conditions that must be met before mating can occur are checked at different levels of execution. The highest level is the `allMateHebbTime` function, which receives the entire population (from one `Bin`) as input. It finds animals that may be able to mate with each other and passes them on to the `mateHebbTime` function, which takes two parent animals as input and returns the parents along with two offspring as output if mating is successful. Animals are only allowed one action per time step. This means they can only mate once per time step, and therefore once per function call to `allMateHebbTime`.

For a potential mating pair to be sent from `allMateHebbTime` to `mateHebbTime`, they must both have reached their respective maturity ages, have mating countdowns equal to zero, be within close proximity of each other, be of the same species and be of opposite sexes. The function that tests whether two sexes are opposite is `differentSex`. It is essentially a Boolean XOR function. The inputs are the Boolean representations of the sexes, and `True` is returned only if the two values are different.

Inputs:

- two sexes as Boolean variables

```
>differentSex ::  Bool → Bool → Bool
>differentSex a b = a ≠ b
```

If a potential mating pair passes all of the tests posed by `allMateHebbTime`, they must still pass the final tests in `mateHebbTime`. This is something of a point of no return for the potential parents, because if they are found to be unable to mate at this point then they have still squandered their action on the mating attempt. This function checks to see whether both animals have more energy than their respective mating costs, and also to see if both animals have the same mating time trait value. Mating fails if these tests are not passed.

The `mateHebbTime` function has two cases. The first case is for when the mating cost and the mating time tests both pass, and mating occurs. The second case is for when mating does not occur. In this case the two parent animals are returned unchanged without any offspring. Their last actions are set to 6 for failed mating. In the first case, where mating occurs, the two parents are returned, having lost the energy expended in mating, along with the offspring they created. The two parents have their last action values set to 1 for mating. The offspring are created using the `newHebbTimeAnimal` function. They are assigned chromosomes derived from the command `mateGenes` from the *Genetic Algorithms* section. Inputs to `mateHebbTime` provide positions within the chromosome to perform crossover, as well as all of the data needed for mutation, in case it occurs.

The starting energy of the first offspring is derived from the first parent, and the starting energy of the second offspring is derived from the second parent. The amount is equal to the parent's start energy trait multiplied by the amount of energy the parent expended in mating, namely its own mating cost.

Both offspring have an initial position equal to the position of the first parent and a heading equal to that of the second parent. Because mutation may have occured, the `restrictTrait` and `restrictTimeDelay` functions are mapped across the corresponding chromosomes. No such function call is needed for the weights chromosome because its values are allowed to mutate unrestricted.

The animals are returned in a list due to the uncertainty as to how many animals will be returned. When mating fails, two animals are returned, but when it succeeds, four animals are returned.

Inputs:

- tuple of two ID numbers
- tuple of two parent `Animal`'s
- tuples of determining factors for both offspring for each of three chromosomes
- tuples of positions to perform mutation for both offspring for each of three chromosomes
- tuple of crossover points for the three chromosomes
- tuples of mutation modifications for both offspring for each of three chromosomes

```
>mateHebbTime ::  (Integer,Integer) → (Animal,Animal)
>    → ((Float,Float),(Float,Float),(Float,Float))→ ((Int,Int),(Int,Int),(Int,Int))
>    → (Int,Int,Int) → ((Float,Float),(Float,Float),(Int,Int)) → [Animal]
>mateHebbTime ($i_1$,$i_2$) (m,f) ($c_T$, $c_W$, $c_D$) ($m_T$, $m_W$, $m_D$) ($cr_T$, $cr_W$, $cr_D$) ($n_T$, $n_W$, $n_D$)
> | matT ∧ enrT =
>    [m {energy = $en_M$ - $mc_M$, lastAction = 1, matingCountdown = $mr_M$},
>    f {energy = $en_F$ - $mc_F$, lastAction = 1, matingCountdown = $mr_F$},
>    newHebbTimeAnimal
>       ($mc_M$×$se_M$) $i_1$ pos hed (map restrictTrait $n_{T_1}$) $n_{W_1}$ (map restrictTimeDelay $n_{D_1}$) ins,
>    newHebbTimeAnimal
>       ($mc_F$×$se_F$) $i_2$ pos hed (map restrictTrait $n_{T_2}$) $n_{W_2}$ (map restrictTimeDelay $n_{D_2}$) ins]
> | otherwise = [m {lastAction = 6}, f {lastAction = 6}]
>    where
>       enrT = $en_M$ > $mc_M$ ∧ $en_F$ > $mc_F$
>       matT = (matingTime $tr_M$) ≡ (matingTime $tr_F$)
>       $se_M$ = take01toFloat (head $tg_M$) minStartEn maxStartEn
>       $se_F$ = take01toFloat (head $tg_F$) minStartEn maxStartEn
>       $mr_M$ = matingRecovery $tr_M$
>       $mr_F$ = matingRecovery $tr_F$
>       pos = position m
>       hed = heading f
>       $en_M$ = energy m
>       $en_F$ = energy f
>       $tr_M$ = traits m
>       $tr_F$ = traits f
>       $mc_M$ = mateCost $tr_M$
>       $mc_F$ = mateCost $tr_F$
>       $tg_M$ = traitGenes $tr_M$
>       $tg_F$ = traitGenes $tr_F$
>       $mu_M$ = mutationRate $tr_M$
>       $mu_F$ = mutationRate $tr_F$
>       $bo_M$ = body m
>       $bo_F$ = body f
>       $wg_M$ = weightGenes $bo_M$
>       $wg_F$ = weightGenes $bo_F$
>       $td_M$ = delayGenes $bo_M$
>         tdF = delayGenes boF
>         ins = getNodeList (hebbTimeNet boM)
>         (nT1,nT2) = mateGenes (tgM,tgF) (muM,muF) cT mT crT nT
>         (nW1,nW2) = mateGenes (wgM,wgF) (muM,muF) cW mW crW nW
>         (nD1,nD2) = mateGenes (tdM,tdF) (muM,muF) cD mD crD nD
```

The only function that calls mateHebbTime is allMateHebbTime. Together they perform all the checks assuring that the preconditions for mating have been met, but neither of the two performs all checks. Without allMateHebbTime, the mateHebbTime function is incomplete. The reverse is also true since mateHebbTime is called from allMateHebbTime. The two are inextricably linked.

The allMateHebbTime function takes as input all of the animals from a single Bin. It goes through this list one animal at a time recursively. When dead animals are encountered they are passed over because, for obvious reasons, dead animals cannot mate. Otherwise the function runs until the list of animals is empty. When living animals are under consideration, checks are done to see if mating may be possible. The mating countdown and age, in comparison to maturity age, are checked first because these checks are the quickest and simplest to perform. The third check looks simple, but is actually quite complex. It is a check to see if the list close is not empty. If this is the case then mateHebbTime is called to attempt mating.

The contents of close are generated with a list comprehension in the where clause of the function. It is a list of living animals that are potential mates for the current animal of interest. A potential mate is defined as one that passes several tests. Checks of the mating countdown and maturity, which the animal of interest has already passed, are the first tests performed within the list comprehension. The list comprehension restricts itself to animals of the same species and opposite sex as the animal of interest. They must also be within a close proximity to this animal. Being within a close proximity means that the distance between the two animals is at most the radius of the smaller, added to interactionDistance. If this list is not empty, then it is likely to at least be small. Whichever animal ends up at the head of the list is taken as the mate.

When all of the tests are passed and a potential mate is found, it is passed to mateHebbTime with all of the necessary data. This data is disentangled from lists of input to allMateHebbTime and cast into the necessary formats for mateHebbTime by a series of anonymous functions at the end of the where clause. In the recursive call to allMateHebbTime the animal chosen as the animal of interest's mate must be removed from the list of animals, because each animal can only mate once per time step. This is accomplished by the remove function defined later in the *Standard Code Extensions* section.

At the end of the function's execution, a tuple of two lists of animals is returned. The tuple is constructed as the recursive calls collapse after reaching the base case. Each of the recursive cases has an anonymous function that assigns animals to the proper list of the two in the tuple. When the tuple is returned, the first list contains all animals that are still free to perform an action on the current time step and the second list contains all the rest. Dead animals are assigned to the second list. An animal for which there are no potential mates, or that is not able to mate at the moment, is placed in the first list. If the mateHebbTime function is called, then the animals it returns are placed in the second list. When mating is successful this makes perfect sense, because the animals returned by the function are either newborns or have just mated, and can therefore not act again on the current time step. However, if mating fails at the mateHebbTime level, why cannot the parents perform some other action on the current time step? The answer is that the two animals wasted their action on a failed attempt at mating. Real world examples for such an occurrence include a bird that fails to sufficiently impress a potential mate with its mating dance and a potential mother whose body is too weak or undernourished to support a growing fetus. Perhaps the sperm was unable to fertilize the egg. For whatever reason, these animals have squandered their time on an unsuccessful mating attempt, and are not allowed to act again on the current time step.

Inputs:

- list of ID numbers
- list of animals
- list of determining factors for mutation
- list of random positions in the traits chromosome

- list of random positions within the weights and time delays chromosomes
- list of random `Float` type mutations
- list of random `Int` type mutations

```
>allMateHebbTime ::  [Integer] → [Animal]
>    → [Float] → [Int] → [Int] → [Float] → [Int] → ([Animal],[Animal])
>allMateHebbTime _ [] _ _ _ _ _ = ([],[])
>allMateHebbTime is ((Dead i p e la):fs) cs $p_Ts$ $p_Ns$ $n_Fs$ $n_Is$
> = (λ b (as,bs) → (as,b:bs)) (Dead i p e la) (allMateHebbTime is fs cs $p_Ts$ $p_Ns$ $n_Fs$ $n_Is$)
>allMateHebbTime is (f:fs) cs $p_Ts$ $p_Ns$ $n_Fs$ $n_Is$
>    | matingCountdown f > 0 ∨ age f ≤ maturity myTr ∨ close ≡ []
>       = (λ a (as,bs) → (a:as,bs)) f (allMateHebbTime is fs cs $p_Ts$ $p_Ns$ $n_Fs$ $n_Is$)
>    | otherwise =
>       (λ b (as,bs) → (as,b++bs))
>       (mateHebbTime
>          i (f, part) c m cr n) (allMateHebbTime nis rest ncs $np_Ts$ $np_Ns$ $nn_Ds$ $nn_Is$)
>       where
>          myPos = position f
>          mySex = sex myTr
>          myTr = traits f

>          close =
>          [ o | o ← (livingAnimals fs), differentSex mySex (sex (traits o)),
>          speciesEq myTr (traits o), (matingCountdown o) ≡ 0, age o > maturity (traits o),
>          distance myPos (position o)
>             ≤ (min (radius myTr) (radius (traits o)))+interactionDistance]

>          part = head close
>          rest = fst (remove part fs)

>          (i, nis) = (λ ($i_1$:$i_2$:is) → (($i_1$,$i_2$),is)) is
>          (c, ncs) = (λ ($c_{T_1}$:$c_{T_2}$:$c_{W_1}$:$c_{W_2}$:$c_{D_1}$:$c_{D_2}$:cs)
>             → ((($c_{T_1}$,$c_{T_2}$),($c_{W_1}$,$c_{W_2}$),($c_{D_1}$,$c_{D_2}$)),cs)) cs
>          (m,cr,$np_Ts$,$np_Ns$) =
>             (λ ($m_{T_1}$:$m_{T_2}$:$cr_T$:$p_Ts$) ($m_{W_1}$:$m_{W_2}$:$cr_W$:$m_{D_1}$:$m_{D_2}$:$cr_D$:$p_Ns$)
>                → ((($m_{T_1}$,$m_{T_2}$),($m_{W_1}$,$m_{W_2}$),($m_{D_1}$,$m_{D_2}$)),($cr_T$,$cr_W$,$cr_D$),$p_Ts$,$p_Ns$)) $p_Ts$ $p_Ns$
>          (n,$nn_Ds$,$nn_Is$) =
>             (λ ($n_{W_1}$:$n_{W_2}$:$n_{T_1}$:$n_{T_2}$:$n_Fs$) ($n_{I_1}$:$n_{I_2}$:$n_Is$)
>                → ((($n_{W_1}$,$n_{W_2}$),($n_{T_1}$,$n_{T_2}$),($n_{I_1}$,$n_{I_2}$)),$n_Fs$,$n_Is$)) $n_Fs$ $n_Is$
```

Individual Movement

More basic than the act of mating is the act of moving. On every time step each animal turns and moves forward according to the inputs it receives to its neural network. The turn can be positive or negative, which correspond to left and right turns. The maximum number of radians that the animal can turn in either direction is controlled by its maneuverability and size. Once the new heading of the animal is set, its position is updated with a bit of trigonometry. The `movePoint` function accomplishes this.

Given the distance that the animal is to move as well as its updated heading, the function treats the line from the animal to the point it wants to reach as the hypotenuse of a right triangle. This means that the x-coordinate of the destination point is equal to the current x-coordinate added to the cosine of the heading multiplied by the length of the hypotenuse. Similarly, the new y-coordinate is the original y-coordinate added to the sine of the heading multiplied by the hypotenuse.

Inputs:

- position of the `Animal`
- heading of the `Animal` in radians
- distance the `Animal` wants to move

```
>movePoint ::  (Float,Float) → Float → Float → (Float,Float)
>movePoint (x,y) h d = (x + (d × cos h), y + (d × sin h))
```

As mentioned above, an animal turns and moves according to its neural network. The neural network has two outputs: one for turning and one for moving. Incidentally, this means that the last number in any network architecture list must be 2. Activation functions as described earlier map either to the range [0,1] ("positive") or [-1,1] ("full"). The functions in this module are designed to only work properly with positive activation functions. This restricts the two outputs of the neural network to the range [0,1]. Since this is the same range as a trait gene, the `take01toFloat` function can be used to map the turn and movement amounts to their proper ranges. The minimum movement required is `minMove`, which is greater than zero. The maximum movement allowed is determined by the animal's maximum speed trait. The restriction on turning is a bit more complicated. Both higher maneuverability and smaller size increase turning range. The maximum turn that an animal can make in either direction is assigned to `maxMan`, which is $m/(r \times c)$, where m is the animal's maneuverability, r is the animal's radius and c is `maneuverabilityRestriction`. The turning range is then [-`maxMan`, `maxMan`].

The value `maxMan` is defined in the `where` clause of the `inputToActionHebbTime` function, which determines an animal's movement based on inputs to its neural network (how these inputs are obtained is explained below). The function takes an animal as input and returns it with updated heading and position data members. Because the neural networks in this simulation support Hebbian learning and time delays, the animal's neural network is also updated. Because the animal moved, it expended energy. The amount of energy removed is d × r × `moveCostMultiple`, where d is the distance moved and r is the animal's radius. Changes to the animal's age and mating countdown are also taken care of at this stage of execution.

The `inputToActionHebbTime` function works by first using `calcNet` to get the movement and heading changes, as well as the updated neural network. The movement and heading changes are mapped to their proper ranges by `take01toFloat` and then used along with `movePoint` to get the new heading and position of the animal. The animal's energy level is adjusted according to the distance moved. All of these results are used to update the data members of the original animal. Its age is incremented and its mating countdown is decremented (so long as it is greater than zero), and the resulting animal is returned.

This function is only meant for living animals.

Inputs:

- a positive activation function
- a living `Animal`
- a list of inputs to the `Animal`'s neural network

```
>inputToActionHebbTime ::  (Float → Float) → Animal → [Float] → Animal
>inputToActionHebbTime a p is =
> p {position = newPos, heading = newHead, age = (age p) + 1, energy = (energy p) - mCost,
> matingCountdown = max ((matingCountdown p) - 1) 0, body = (body p) {hebbTimeNet = newNet}}
>    where
>       (newNet,[m,t]) = calcNet a (learningRate ts) is (hebbTimeNet (body p))
>       ts = traits p
>       move = take01toFloat m minMove (maxSpeed ts)
>       mCost = move × moveCostMultiple × rad
>       turn = take01toFloat t (-maxMan) maxMan
>       newHead = confineHeading (turn + (heading p))
>        newPos = movePoint (position p) newHead move
>        maxMan = (maneuverability ts) / (maneuverabilityRestriction × rad)
>        rad = radius (traits p)
```

Senses

In the above function an animal moves according to inputs sent to its neural network. These inputs are gathered from the environment and represent the things that an animal senses. The only objects in the environment are plants and other animals. Depending on an animal's diet, species and sex, these two types of objects can be perceived in many different ways. The different categories into which all objects are classified by an animal's perceptions are threat, mate, same species, food, benign and plant. This classification scheme allows some objects to be double or even triple classified. For example, to a cannibal a member of the same species but opposite sex may be classified as mate, food and same species. If the other animal happens to be bigger, then the threat classification would replace the food classification. The meanings of the different classifications are explained in detail below:

Threat A threat to an animal is any other animal that could possibly eat it. To be able to eat another animal, an animal must be at least as big as that animal. If both animals are of the same species, then the threatening animal must be a cannibal. If both animals are of different species, then the threatening animal must be a carnivore.

Mate All of the conditions required for two animals to be able to mate are thoroughly described above, but not all of these conditions are checked at this stage of classification. Until the animals are right next to each other they can only get a limited sense of which animals are potential mates. In order to be classified as a mate, an animal must be of the same species and different sex as the animal doing the classification. Logically, any animal that is classified as a mate is also classified as a member of the same species.

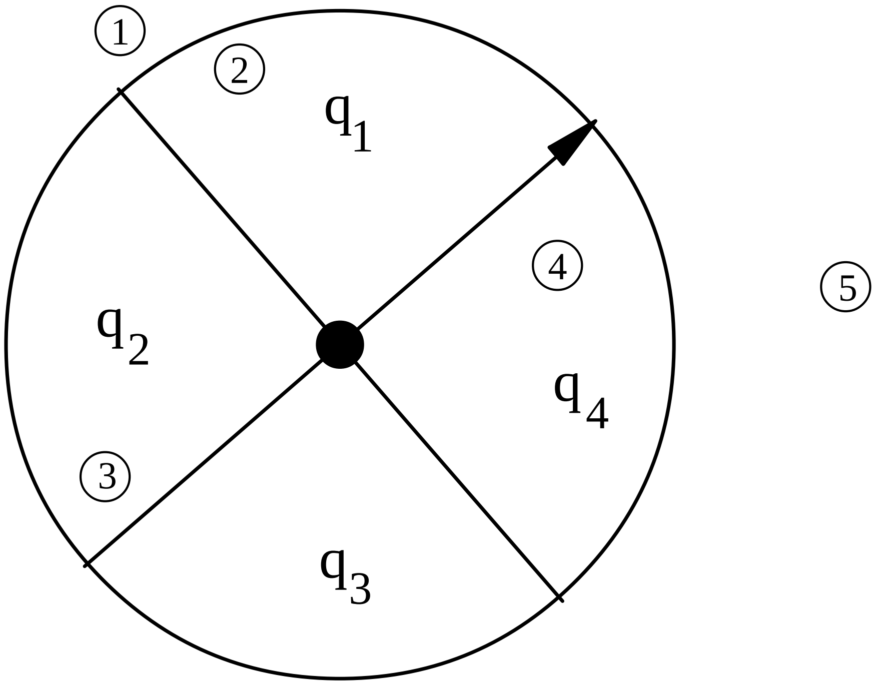
Same Species This one is self explanatory. If another animal is the same species as the one that perceives it, then it is classified as such.

Food Food is any object that the animal can eat. If the animal is a herbivore this includes plants. If it is a cannibal or a carnivore this can include other living animals. For one living animal to eat another means that the one being eaten is of smaller or equal size. Cannibals can eat smaller living animals of the same species and carnivores can eat smaller living animals of other species. If the animal is a carrion eater then it recognizes corpses as a food source. Size difference is not an issue when recognizing carrion as a food source (technically, dead animals no longer have a size trait).

Benign If an animal is not classified in any of the above categories then it is classified as a benign animal. To make it clearer, a benign living animal must be of a different species and can be neither food nor threat (being of a different species automatically excludes it from being a mate). All dead animals are benign as long as the animal collecting data is not a carrion eater.

Plant Regardless of whether the animal can eat them or not, every animal takes notice of the plants in the environment.

The objects that an animal can sense are limited by its sight trait. It only senses objects whose distances from it are at most its sight trait. This means the animal can sense objects within a circle with radius equal to its sight with itself at the center. This circle is divided into equal size quadrants along the animal's line of orientation (its heading) and its perpendicular. These four slices of the circle are the four regions into which an animal further sorts the objects it senses. Only two aspects are important when an animal sorts the objects it senses: the object's classification as defined above and the quadrant, relative to the sensing animal's heading, that the sensed animal occupies.

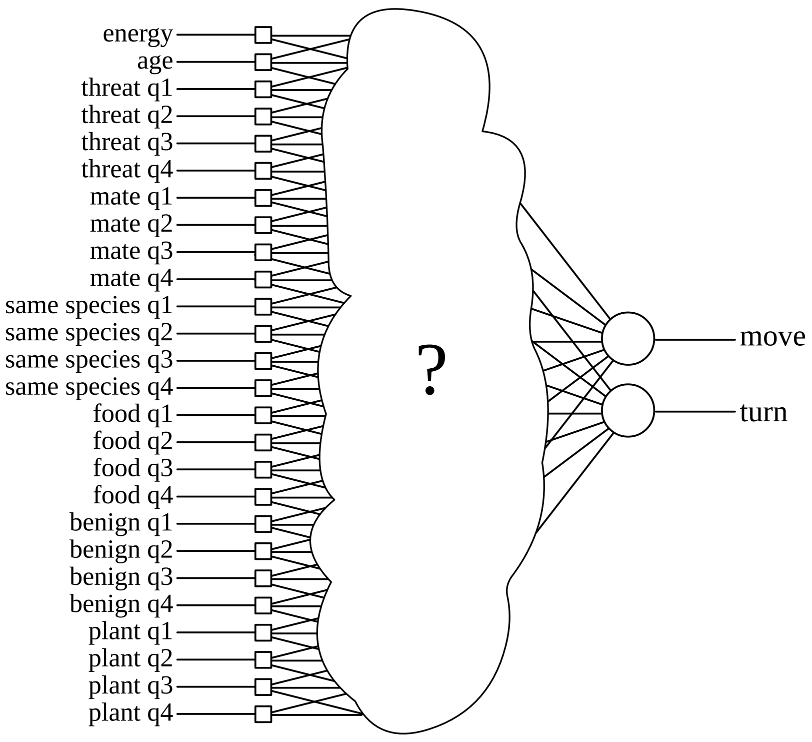


The black dot is an `Animal` *with its heading and four surrounding quadrants shown. The disk around it is represents its range of sight. The numbered white dots are other objects of interest in the environment. Objects 1 and 5 are outside the of the* `Animal`*'s range of sight, so they are not sensed by it. Object 2 is in quadrant 1, object 3 is in quadrant 2 and object 4 is in quadrant 4. Quadrant 3 holds no objects to be sensed.*

This means of using a neural network to sense objects within the environment is similar to the method used by Ward, Gobet and Kendall in "Evolving Collective Behavior in an Artificial Ecology" [10]. They modeled artificial fish using neural networks that sensed predators, prey and food stuffs. They also divided the area surrounding the sensing animal into four quadrants, but in addition to this there are two smaller sense regions on each side of the fish, which represent their "lateral line" system. These extra sense regions are excluded from this project because the number of inputs to an animal's neural network is already immense.

The data that an animal collects from its external environment is represented by 24 values divided into 6 tuples of 4 values each. Each tuple corresponds to one of the 6 object classifications above and the 4 values in each tuple are for each of the four quadrants surrounding the animal. The values themselves are simply counts of the number of each type of animal sensed in each quadrant. For example, there is a count for the number of threats that are in front of and to the left of the animal's heading, and also a count for the number of plants behind and to the right of the animal. The sum of the 4 values in one tuple are all the objects of a given type surrounding the animal (within its sight range). Because the quadrants are disjoint there is no double counting of objects within one tuple, but some objects are counted in more than one tuple (when they satisfy multiple classifications as described above).

Two more senses that animals have provide a limited awareness of their own mortality. In addition to the 24 external senses, each animal also knows its own energy level and age. All together this makes for a list of 26 values, which when arranged in a list make up the series of inputs to an animal's neural network, meaning that the first value in any network architecture list must be 26. The neural network then extrapolates two output values from these 26 inputs.



All `Animal`'s have a neural network that accepts the 26 inputs and creates the two outputs shown above. The number and size of the hidden layers in between is set by the network architecture, which is the same for all `Animal`'s. The number after each 'q' indicates which quadrant an input comes from.

The function that calculates what these inputs are is `gatherInputs`. It is a very complex function, and it only returns the input list for a single animal. It runs in linear time since the one animal in question must access information about all others. Because every animal must gather inputs for its neural network, it takes quadratic time for all input lists to be generated for the whole population.

Data about the number and relative locations of plants is generated by the separate function `visiblePlants` from the section *Plants*, and then added into the final output before it gets returned. The function `gatherInputs` has two cases. If the animal gathering data is a herbivore then the values returned by `visiblePlants` are counted as both food and plants. Otherwise they are only counted as plants. All information about other animals is collected by the helper function `go`, which performs an extensive case analysis in order to classify animals.

The `go` function uses a large and unwieldy accumulator. It is a 5-tuple of 4-tuples, providing for 20 of the 24 external inputs. The other 4 are generated by `visiblePlants`. The function traverses the list of other animals recursively while maintaining the accumulator, and returns it when the list is empty.

The `go` function works by first pattern matching for dead animals. They are easier to deal with, and having this case precede the living animal case saves computation. The second pattern match can then safely assume a living animal. The first case check of both of these patterns assures that the animal in question is within sight range. If it is not, then it is ignored and the recursion continues. Otherwise the remaining case checks must be carried out.

If the animal is dead, then it is only of interest to carrion eaters as a food source, so this case is easy to deal with. If the animal gathering the data is a carrion eater then the dead animal is added as a food object. Otherwise it is considered benign. In either case the accumulator is updated by the helper function `toQ`, which in conjunction with `relativeQuadrant`, from the section *Environment*, increments the value for the correct quadrant within the proper tuple. All of the cases in `go` work more or less in this way. The case checks determine which categories an animal belongs to and `relativeQuadrant` determines which quadrant the animal is in. Then `toQ` is used to apply this information and increment the proper counters within the accumulator.

There are many cases to check when the animal being examined is living. All of the tests done in the case checks are done in the `where` clause of either `go` or `gatherInputs` so as to cut back on repetitive computation (making use of Haskell's memoization capabilities). The cases are ordered so as to make some checks unnecessary on lower cases. This works because the outcome of certain tests can be determined by virtue of the fact that execution fell through an earlier case. In spite of this there are still many cases to check.

The first check after the sight range check catches the peculiar case of an animal that is threat, mate, same species, and food. This can only happen when both animals are of the exact same size, and both are cannibals of the same species but opposite sex. The next case performs the same checks except for the opposite sex check, which one can assume to be `False` if the other three are `True`. This catches the case of an animal that is a threat, food and of the same species, but not a mate.

The next two checks use the same trick with the opposite sex check. The first check catches the case of a member of the same species that is both mate and food. To see that the animal is small enough to eat, there is a check that tests whether it is at most as big as the animal collecting data. If it is the same size, then it cannot be a cannibal, and therefore a threat, because that case was caught by the previous check. If it is a cannibal it must be smaller in size, and therefore not a threat. This is why no check is made to see if the animal is a cannibal. Only the animal collecting the data needs to be a cannibal. The second check is the same, except it lacks the opposite sex check and the animal turns out not being a mate.

The next two checks are similar, but instead of classifying the animal as food it classifies the animal as a threat. The checks for whether or not the animal collecting data was a cannibal are replaced with checks to see if the animal being sensed is a cannibal, and the checks to see if the sensed animal is smaller are replaced with checks to see if the sensed animal is bigger. The two checks work similarly and for the same reasons.

By the next check any threats or food of the same species have been caught and processed. If the sensed animal is of the same species at this point, then the only other classification it might fulfill is that of mate. That case is caught by this check, and should the animal not be a mate then the next check catches any animal that is solely classified as an animal of the same species. Since all members of the same species have been caught by this point, it is safe to assume an animal is of a different species and forego the species check. Since mates must be of the same species, this also means that the opposite sex check no longer needs to be performed.

The next case is for when both the sensing and sensed animals are carnivores of the same size. Since they are of different species, this makes the sensed animal both a threat and food to the sensing animal. The next two cases catch animals that are food and threat respectively. They use the same trick of replacing a smaller size check with a bigger size check as the cannibal threat and food checks above. What in the first check was a test of whether the sensing animal is a carnivore becomes in the second check a test of whether the sensed animal is a carnivore.

Should execution run through all of these cases then the animal must be benign.

The following table summarizes the case analysis presented above. It lists the different checks that are performed when an animal gathers inputs from its environment. They are in the same order as they appear in the function below. The top row of the first six columns list Boolean values that are the results of predicates: `specT` is true if both animals are of the same species, `cann` if the animal gathering inputs is a cannibal, `cannO` if the animal being sensed is a cannibal, `carn` if the animal gathering inputs is a carnivore, `carnO` if the animal being sensed is a carnivore, `sexT` if both animals are of opposite sex. For each check, a value of 1 means the value is true and 0 indicates false. If instead the value name is shown, then it means that its value cannot affect how the sensed animal will be classified. When a 0 is followed by an *, it means that the value was not directly checked, but rather inferred given the results of previous checks. Otherwise there is a direct check performed for that case in the function. The size column lists what sizes pass the check, and the classification column tells how the sensed animal is classified.

specT	cann	cann0	carn	carn0	sexT	size	classification
1	1	1	carn	carn0	1	likeSize	threat, mate, same species, food
1	1	1	carn	carn0	0*	likeSize	threat, same species, food
1	1	cann0	carn	carn0	1	small ∧ likeSize	mate, same species, food
1	1	cann0	carn	carn0	0*	small ∧ likeSize	same species, food
1	cann	1	carn	carn0	1	big ∧ likeSize	threat, mate, same species
1	cann	1	carn	carn0	0*	big ∧ likeSize	threat, same species
1	cann	cann0	carn	carn0	1	any	mate, same species
1	cann	cann0	carn	carn0	0*	any	same species
0*	cann	cann0	1	1	sexT	likeSize	threat, food
0*	cann	cann0	1	carn0	sexT	small ∧ likeSize	food
0*	cann	cann0	carn	1	sexT	big ∧ likeSize	threat
0*	cann	cann0	carn	carn0	sexT	any	benign

Summary of case analysis in the `gatherInputs` function.

Inputs:

- the sensing `Animal`
- the `PlantGrid` in the sensing `Animal`'s `Bin`
- list of other `Animal`'s in the same `Bin` as the sensing `Animal`

```
>gatherInputs ::  Animal → PlantGrid → [Animal] → [Float]
>gatherInputs f g os
> | herb =
>    [energy f, convert (age f), t1, t2, t3, t4, m1, m2, m3, m4, s1, s2, s3, s4,
>    e1 + p1, e2 + p2, e3 + p3, e4 + p4, b1, b2, b3, b4, p1, p2, p3, p4]
> | otherwise =
>    [energy f, convert (age f), t1, t2, t3, t4, m1, m2, m3, m4, s1, s2, s3, s4,
>    e1, e2, e3, e4, b1, b2, b3, b4, p1, p2, p3, p4]
>    where
>       (herb,carn,cann,carr) = diet trF
>       trF = traits f
>       mySize = radius trF
>       mySight = sight trF
>       myPos = position f
>       myHeading = heading f
>       mySex = sex trF
>       nS = (0.0,0.0,0.0,0.0)

>       [p1, p2, p3, p4] = visiblePlants g myPos myHeading mySight

>       toQ ::  (Float,Float,Float,Float) → Int → (Float,Float,Float,Float)
>       toQ (q1, q2, q3, q4) 1 = (q1 + 1, q2, q3, q4)
>       toQ (q1, q2, q3, q4) 2 = (q1, q2 + 1, q3, q4)
>       toQ (q1, q2, q3, q4) 3 = (q1, q2, q3 + 1, q4)
>       toQ (q1, q2, q3, q4) 4 = (q1, q2, q3, q4 + 1)
>       toQ (q1, q2, q3, q4) _ = (q1, q2, q3, q4)

>       ((t1, t2, t3, t4),(m1, m2, m3, m4),(s1, s2, s3, s4),(e1, e2, e3, e4),(b1, b2, b3, b4))
>          = go os (nS, nS, nS, nS, nS)

>       go [] is = is
>       go ((Dead _ p _ _):os) (t,m,s,e,b)
>          | distance myPos p > mySight = go os (t,m,s,e,b)
>           | carr = go os (t,m,s,toQ e qO,b)
>           | otherwise = go os (t,m,s,e,toQ b qO)
>             where qO = relativeQuadrant myPos myHeading p
>       go (o:os) (t,m,s,e,b)
>           | distance myPos pO > mySight = go os (t,m,s,e,b)
>           | specT ∧ cann ∧ cannO ∧ likeSize ∧ sexT
>             = go os (toQ t qO,toQ m qO,toQ s qO,toQ e qO,b)
>           | specT ∧ cann ∧ cannO ∧ likeSize
>             = go os (toQ t qO,m,toQ s qO,toQ e qO,b)
>           | specT ∧ cann ∧ (small ∨ likeSize) ∧ sexT
>             = go os (t,toQ m qO,toQ s qO,toQ e qO,b)
>           | specT ∧ cann ∧ (small ∨ likeSize)
>             = go os (t,m,toQ s qO,toQ e qO,b)
>           | specT ∧ cannO ∧ (big ∨ likeSize) ∧ sexT
>             = go os (toQ t qO,toQ m qO,toQ s qO,e,b)
>           | specT ∧ cannO ∧ (big ∨ likeSize)
>             = go os (toQ t qO,m,toQ s qO,e,b)
>           | specT ∧ sexT
>             = go os (t,toQ m qO,toQ s qO,e,b)
>           | specT
>             = go os (t,m,toQ s qO,e,b)
>           | carn ∧ carnO ∧ likeSize
>             = go os (toQ t qO,m,s,toQ e qO,b)
>           | carn ∧ (small ∨ likeSize)
>             = go os (t,m,s,toQ e qO,b)
>           | carnO ∧ (big ∨ likeSize)
>             = go os (toQ t qO,m,s,e,b)
>           | otherwise
>             = go os (t,m,s,e,toQ b qO)
>               where
>                 pO = position o
>                 qO = relativeQuadrant myPos myHeading pO
>                 trO = traits o
>                 specT = speciesEq trF trO
>                 (_, carnO, cannO,_) = diet trO
>                 rad = radius trO
>                 big = mySize < rad
>                 likeSize = mySize ≡ rad
>                 small = mySize > rad
>                 sexT = differentSex mySex (sex trO)
```

Group Movement

The last two functions combined allow for a single animal to collect inputs and then make its movement for the time step. The next function combines the two to make the entire population gather inputs and move. The `allMove` function goes through a list of animals recursively, having each one gather inputs with `gatherInputs`. The results are then sent to `inputToActionHebbTime` along with the animal. All living animals move at the beginning of each time step. Afterwards they have a chance to perform one of several available actions (mating, eating). It is also possible that no actions are available to the animal after its move, in which case it does nothing on the time step. For this reason the last action data member is set to 0 for nothing at this stage of execution. If the animal does in fact do something on the time step then this value is changed to the appropriate value, but otherwise it stays as nothing to reflect what the animal did (or more properly, did not do).

One interesting note about the recursion performed by the `go` helper function is that a counter is used, because the result of every step, the animal that moved, is appended to the end of the list of current animals.

If the animal was not placed back in the list, then the next animal to gather inputs would not be able to sense it. However, since the animal is put back in the list after moving, this means that the animal movements are not simultaneous. Animals further on in the list can sense where the earlier animals have moved to before determining their own action. This is completely dependent on order within the list and has nothing to do with an animal's traits.

Inputs:

- a positive neural network activation function
- a `PlantGrid`
- the list of `Animal`'s from the same `Bin` as the `PlantGrid`

```
>allMove ::  (Float → Float) → PlantGrid → [Animal] → [Animal]
>allMove a g fs = go (length fs) fs
> where
>    go ::  Int → [Animal] → [Animal]
>    go 0 fs = []
>    go n (f:fs) = (moved {lastAction = 0}):(go (n - 1) (fs++[f]))
>       where
>          moved = inputToActionHebbTime a f (gatherInputs f g fs)
```

Eating

After mating and moving, the next important action that animals can perform is eating. Eating is a collection of several actions because there are so many things to eat. Animals can eat plants, corpses, members of their own species or members of other species. Each of these cases needs to be handled slightly differently. The simplest of these cases is for eating plants, and since this case can be handled separately from the others it is dealt with first.

The `allEatPlants` function takes a `PlantGrid` and a list of animals as input and has the animals eat the plants, if they are able to. The list of animals must consist of only living animals in order to prevent errors, and these animals should only be ones that have yet to act on the current time step (the function itself does not assure this). Along with an updated `PlantGrid`, the function returns two lists of animals: those that ate a plant and those that are still free to act.

As with many other functions in this module, `allEatPlants` uses a recursive helper function named `go` that maintains an accumulator, which gets returned when `go` reaches its base case. The accumulator contains a list of changes to be made to the `PlantGrid` as well as two lists of animals: ones that ate and ones that did not. Once the accumulator is returned the `PlantGrid` updates are applied with the `massUpdate2D` function.

When checking to see if an animal can eat a plant, its position in the continuous world is converted to one in the discrete world of the `PlantGrid` with `coordToArray`. Because the animal will have moved before getting the chance to eat a plant, it is possible that the animal moved outside of the bounds of its `Bin`. The *Bins* section explains how such rogue animals are put in their proper place at the end of the time step. However, being outside of the `Bin` means the animal is also outside of the `PlantGrid`. If this is the case, then its supposed position within the `PlantGrid` is actually out of bounds. The first test case of `go` checks this situation and does not allow the animal to eat if this is the case. The animal is put in the list of animals that did not eat and the recursion continues.

The next case poses three tests that must be passed for the animal to eat a plant. First of all, the animal must have the herbivore diet trait. Secondly, the plant at the animal's location cannot have already been eaten on the current time step. This test is done with a quick look at the `PlantGrid` updates in the accumulator. If the position of the animal is already being updated to hold `False`, then the plant was already eaten. The last check makes sure that there is a plant at the location to be eaten. This is done by looking up the contents of the position with the `BRAMatrix` function `lookup2D`. If all three tests pass then the animal eats the plant. An entry is added to the accumulator's `PlantGrid` updates and the animal is added to the list of those that ate. Its last action is set to 2 for eating a plant and its energy is increased by `plantNutrition`. If one or more of the tests fail then the animal does not eat the plant and is added to the list of animals that are still free to act.

Inputs:

- a PlantGrid
- a list of living Animal's that have not yet acted on the current time step

```
>allEatPlants :: PlantGrid → [Animal] → (PlantGrid, ([Animal],[Animal]))
>allEatPlants g fs = (massUpdate2D g eaten, fed)
> where
>    siz = matrixSize g
>    (eaten, fed) = go fs ([],([],[]))

>    go :: [Animal] → ([((Int,Int),Bool)], ([Animal],[Animal]))
>       → ([((Int,Int),Bool)], ([Animal],[Animal]))
>    go [] (ps,os) = (ps, os)
>    go (f:fs) (ps,(ns,os))
>       | out = go fs (ps,(f:ns,os))
>       | h ∧ not (elem (pos,False) ps) ∧ lookup2D pos g
>          = go fs ((pos,False):ps, (ns, an:os))
>       | otherwise = go fs (ps,(f:ns,os))
>          where
>             an = f {energy = (energy f) + plantNutrition, lastAction = 2}
>             pos = coordToArray (position f)
>             out = fst pos ≥ fst siz ∨ fst pos < 0 ∨ snd pos ≥ snd siz ∨ snd pos < 0
>             (h, _, _, _) = diet (traits f)
```

Before getting to the allEatAnimals function, which handles carnivores, cannibals and carrion eaters, two small functions are defined. Both are used by allEatAnimals and are quite simple, but they only work properly when called from allEatAnimals.

The first is the eatCarrion function, which has a carrion eater eat a corpse. It is possible that the dead animal's energy drops to zero as a result of being eaten, in which case it is removed from the environment. Otherwise it remains. In this case both the carrion eater and the dead animal are returned in a list with the dead animal second. If the dead animal is removed from the environment then the carrion eater is returned in the list by itself.

The amount of energy that a carrion eater gets from eating a corpse is its own radius multiplied by the carrionMultiplier, unless the corpse has less energy than that amount. When that happens the carrion eater takes all of the remaining energy and the corpse is not returned by the function. Otherwise the corpse loses the energy that was taken and is returned by the function. In any case the carrion eater gains energy equal to what the corpse loses.

Inputs:

- a living carrion eater
- a corpse

```
>eatCarrion :: Animal → Animal → [Animal]
>eatCarrion f (Dead i p e la)
> | deEn > 0 = [f {energy = (energy f) + enEx}, Dead i p deEn la]
> | otherwise = [f {energy = (energy f) + enEx}]
>    where
>       enEx = min e (carrionMultiplier × (radius (traits f)))
>       deEn = e - enEx
```

The next function is for the situation of one animal eating a living animal. It serves for both the case of a carnivore eating a member of another species and a cannibal eating a member of its own species. The first animal sent to the function is the predator, which is either a carnivore or cannibal (or both). The second animal is the prey. The amount of energy that the predator gains, and that the prey loses, is $(|r_1 - r_2| \times$ `predationMultiplier`$)$ $+$ `predationConstant`, where r_1 is the predator's radius and r_2 is the prey's radius. The difference in radii is used because bigger animals can take bigger bites. If two animals are the same size it is still possible for one to eat the other. In this case the difference in the radii is zero, but the `predationConstant` assures that even in this case the predator gets a bit of energy when it eats. As with the case of carrion eating, this energy exchange amount is only used if the prey animal actually has the energy to spare. Otherwise it only gives away what it has to give.

Should the prey animal's energy drop below its own death point as a result of being eaten then it is converted to a dead animal with -2 for eaten as its last action (note that even if the animal already performed an action on this time step it would be erased by dying). Otherwise it continues to live. In both cases its energy level goes down by the amount that the predator gains. In the returned list of two animals, the predator is first and the prey is second.

Inputs:

- a living predator `Animal`
- a living prey `Animal`

```
>eatAnimal ::  Animal → Animal → [Animal]
>eatAnimal hunt prey
> | dp > nv = [hunt {energy = (energy hunt) + enEx}, Dead (idN prey) (position prey) nv (-2)]
> | otherwise = [hunt {energy = (energy hunt) + enEx}, prey {energy = nv}]
> where
>    nv = pEn - enEx
>    dp = deathPoint prey
>    pEn = energy prey
>    enEx = min pEn maxEx
>    maxEx = (((radius (traits hunt)) - (radius (traits prey))) × predationMultiplier)
>              + predationConstant
```

The `eatCarrion` and `eatAnimal` functions are but a small part of `allEatAnimals`. This function takes as input two lists of animals: those that have already acted on the current time step and those that have yet to act. Animals that have already acted are present only as potential food for those animals that have yet to act. The function can get away with returning a single list of animals because of all the actions that an animal can perform, eating another animal comes last. This means that after this function there are no more chances to act, so it does not matter that animals that acted and those that did not are mixed in the same list together.

Unlike many other large functions in this module, `allEatAnimals` does not need a single massive helper function to do all of its work. It makes recursive calls to itself, and on each step an animal from the "free to act" list is moved to the "already acted" list (either because it acted or because it missed its last chance to act) until all animals are in one list, which gets returned.

Naturally, dead animals that are encountered in the "free to act" list are moved to the "already acted" list and execution continues recursively. Otherwise the function needs to determine if the current animal can eat one of the other animals in one of the two lists. First a list comprehension is used to generate a list of animals that are close enough to the current animal to be eaten. These animals are taken from both input lists, and their distance from the current animal must be less than the radius of the current animal plus `interactionDistance`. Animals in this list must have an energy level greater than zero (if the animal has an energy level of zero or less it means that it is simply waiting to be removed by the `allDie` function defined below). If this list is empty then the current animal is not close enough to any other animal to eat it. The animal has missed its chance to act. It does nothing and is moved to the "already acted" list on the recursive call.

If the list is not empty then execution continues. The next three cases are dependent both on the proximity of food and the current animal's own diet traits. Each case works in the same way, but searches for a different type of food. Three more lists are generated from the list of nearby animals. Each list contains the edible animals for one of the three eating types. One list holds nearby dead animals for carrion eaters, one holds nearby prey of the same species for cannibals and the third holds prey of different species for carnivores. Given a choice between food sources, an animal will choose carrion first, then members of another species and then finally members of the same species (only if the animal has the appropriate diet traits of course). If the animal has the appropriate diet trait and there is food nearby, then the animal chooses to eat the first animal in the list of food sources. The `arrange` helper function makes sure that after the eating action is resolved, every animal is placed in the correct list ("free to act" or "already acted") on the recursive call. The animal that ate always goes into the "already acted" list, because it just ate. The animal that was eaten must be both removed from and returned to the list it started in. Technically, the old version of the animal (before being eaten) is the one removed from the list, and the new version (after being eaten) is inserted back into the same list. The new version is different from the old version because it has lost energy from being eaten. However, if a corpse's energy level dropped to zero, then it is not returned at all. In any case it is the `remove` function from the *Standard Code Extensions* section that removes the old version of the animal from whatever list it was in. Then the new version of the animal, if it exists, is added to that same list.

Before `arrange` can receive its inputs the three lists of food sources must be generated. The list of food sources for carrion eaters is the easiest to generate. The `deadAnimals` function is used on the list of nearby animals. The next two cases are slightly more complicated. Because living prey of any type (same or other species) must be of smaller or equal size to the predator animal, the nearby living animals are first sorted by this criterion using a list comprehension. This list is further divided into prey of the same species and prey of another species. This is done by the `partition` function from the *Standard Code Extensions* section, which is similar to `filter` except that it puts elements that return `False` to its predicate into a second list instead of discarding them. The function returns two lists: animals of the same species and animals of another species. The animals of another species are used as food sources for carnivores, but the list of animals of the same species is filtered once more. Among cannibals, there is a grace period for all newly born animals during which no cannibal will eat them. This is done to discourage cannibals from eating their own offspring directly after giving birth. The grace period is a number of time steps set by `cannibalGracePeriod`. Only animals with ages greater than this value are considered food sources to cannibals. Notice that there is no similar grace period for parents. Since they are born fully formed, offspring can eat their parents as soon as they are born.

Once an animal has decided what it will eat (once one of the case checks succeeds), either `eatCarrion` or `eatAnimal` is called, depending on what is being eaten. Before being sent to one of these functions the animal that is eating has its last action updated to reflect what it is about to eat (carrion, member of same species or member of other species, with last action codes 5, 4 and 3 respectively). This update cannot be performed in the functions themselves because `eatAnimal` is used for two of these cases. As mentioned above, the first animal within the appropriate list of food sources is chosen as the eaten animal for these function calls, only one of which, if any, will actually execute. This animal along with the list of up to two animals returned by either `eatCarrion` or `eatAnimal` make up the inputs to `arrange`. Being a helper function, `arrange` has access to and makes use of the "free to act" and "already acted" lists sent as input to `allEatAnimals`.

The last case for `allEatAnimals` is the default case, in which none of the nearby animals were considered viable food sources. The current animal does nothing and is added to the "already acted" list.

Inputs:

- tuple of a list of `Animal`'s that can still act followed by a list of `Animal`'s that have already acted

```
>allEatAnimals ::  ([Animal],[Animal]) → [Animal]
>allEatAnimals ([],os) = os
>allEatAnimals ((Dead i p e la):ps,os)
> = allEatAnimals (ps,(Dead i p e la):os)
>allEatAnimals (f:ps,os)
> | nei ≡ [] = allEatAnimals (ps,f:os)
> | carr ∧ dead ≠ [] = allEatAnimals (arrange heDe survCarr)
> | carn ∧ prey ≠ [] = allEatAnimals (arrange hePr survPred)
> | cann ∧ same ≠ [] = allEatAnimals (arrange heSa survCann)
> | otherwise = allEatAnimals (ps,f:os)
> where
>    (herb,carn,cann,carr) = diet myTr
>    myTr = traits f
>    myPos = position f
>    mySiz = radius myTr

>    nei =
>       [ o | o ← (ps++os),
>       distance myPos (position o) < (radius myTr) + interactionDistance, energy o > 0]
>    dead = deadAnimals nei
>    small = [ o | o ← livingAnimals nei, radius (traits o) ≤ mySiz]
>    (sameSpe,prey) = partition ((speciesEq myTr).traits) small
>    same = filter (λo → age o > cannibalGracePeriod) sameSpe

>    heDe = head dead
>    hePr = head prey
>    heSa = head same

>    survCarr = eatCarrion (f {lastAction = 5}) heDe
>    survCann = eatAnimal (f {lastAction = 4}) heSa
>    survPred = eatAnimal (f {lastAction = 3}) hePr

>    arrange ::  Animal → [Animal] → ([Animal],[Animal])
>    arrange h ss
>       | inPs = (remPs++(tail ss),(head ss):os)
>       | otherwise = (ps,ss++remOs)
>          where
>             (remPs,inPs) = remove h ps
>             (remOs, _ ) = remove h os
```

Related to eating behaviors is the concept of metabolism. In this simulation, the advantage of having multiple eating behaviors, and thus more ways to obtain energy, comes with the disadvantage of an increased metabolic rate. This means that an animal consumes energy at a higher rate. Specifically, the amount of energy consumed by an animal per time step, in addition to all expenditures from moving, mating and being eaten, is equal to `metabolicRate` multiplied by s^n, where s is the animal's size and n is the number of eating behaviors (diet traits) that the animal possesses. This means that energy costs grow exponentially in relationship to the number of eating behaviors possessed.

The function `metabolism` collects these energy costs at the beginning of each time step, before the animals even move. It checks each animal recursively, calculates the energy cost and subtracts it from the animal's energy. The energy cost is calculated as described above, using the `mRate` helper function to count the number of eating behaviors the animal has.

Inputs:

- a list of `Animal`'s

```
>metabolism :: [Animal] → [Animal]
>metabolism [] = []
>metabolism (o:os) = (o {energy = (energy o) - totCost}):(metabolism os)
> where
>    ts = traits o
>    (a,b,c,d) = diet ts
>    size = radius ts
>    mCost = mRate [a, b, c, d]
>    totCost = metabolicRate × size^mCost

>    mRate :: [Bool] → Int
>    mRate es = length (filter id es)
```

Death

After suffering so much energy depletion from metabolism, moving around and possibly even mating or being eaten, it is possible that some animals have died. When an animal's energy level drops below its death point it is time to die, but the animal is not automatically converted from the `Live` type to the `Dead` type. It remains a living animal with its fatal energy level until it is caught and converted by the `allDie` function. Animals that have outlived their lifespan also remain living until converted by this function.

The call to `allDie` comes at the beginning of each time step, but given that the time steps are executed one after the other, there is little difference between having the function call at the beginning or end of a time step. In addition to converting living animals to dead ones, the function also makes corpses rot and removes them from the environment when their energy levels drop to or below zero. It handles all maintanence operations involving death and dying.

The function takes a list of animals as input and goes through them recursively. Dead animals encountered are dealt with by one of two cases. If the dead animal's energy is not positive, then it is removed from the environment, meaning that it is dropped on the recursive call. If a dead animal still has energy left then it will rot on every time step until it is gone. When an animal rots its energy level is decreased by `decay` and its last action code is set to -3 for rotting. Living animals are dealt with by one of three cases. If an animal's energy level is less than its death point then it dies and has its last action set to -4 for starvation. If its age is greater than its lifespan then it dies and has its last action set to -5 for passing away. Otherwise the animal continues to live and it returned to the list unchanged.

Inputs:

- list of `Animal`'s

```
>allDie :: [Animal] → [Animal]
>allDie [] = []
>allDie ((Dead i p e _ ):fs)
> | e ≤ 0 = allDie fs
> | otherwise = (Dead i p (e - decay) (-3)):(allDie fs)
>allDie (f:fs)
> | energy f < deathPoint f = (Dead (idN f) (position f) (energy f) (-4)):(allDie fs)
> | age f > lifespan (traits f)
>    = (Dead (idN f) (position f) (deathPoint f) (-5)):(allDie fs)
> | otherwise = f:(allDie fs)
```

Energy Restriction

The opposite of an animal having so little energy that it dies is an animal having so much energy that it cannot gain any more. When an animal's energy level surpasses its maximum energy, the `energyCap` function fixes the problem. The function is very simple. Dead animals remain as they were. If a living animal has an energy level higher than its maximum energy then its energy is set equal to its maximum energy. Otherwise it is left as it was.

Inputs:

- list of `Animal`'s

```
>energyCap ::  [Animal] → [Animal]
>energyCap [] = []
>energyCap ((Dead i p e l):os) = (Dead i p e l):(energyCap os)
>energyCap (o:os)
> | eo > me = (o {energy = me}):(energyCap os)
> | otherwise = o:(energyCap os)
>    where
>       eo = energy o
>       me = maxEnergy o
```

Time Step

Each of the functions above deals with one aspect of a single time step. When all of these functions are combined together they make up all of the actions and operations that need to be performed on each time step. The function that combines them all is `masterHebbTimeAction`. Having all actions taken care of by a single function in this module greatly simplifies code in the `Bins` module. The order of actions within a time step is as follows:

1. Living animals pay their metabolism costs.
2. All living animals gather their inputs and make their movement for the time step.
3. These animals are combined with the dead animals and the combined list is sent to `allDie`.
4. The list returned by `allDie` is sent to `allMateHebbTime` to give animals a chance to mate.
5. Animals that are still free to act are given a chance to eat plants by `allEatPlants`. The `PlantGrid` is also updated by this function.
6. Animals that are still free to act after this function have a chance to eat other animals in `allEatAnimals`.
7. The `energyCap` function is used to maintain the maximum energy restriction.

This execution order enforces an action priority for all animals. They can only act once per time step, and certain actions will always be chosen over others when the chance is available. Mating is the most important action. Only if an animal cannot mate will it consider eating. If plants are available to eat, and the animal is a herbivore, it will choose to eat plants over other food sources. Only if neither of these actions were possible will the animal try eating another animal, and according to the `allEatAnimals` function, carrion is the preferred food, followed by animals of other species and then animals of the same species.

The main inputs to `masterHebbTimeAction` are a `PlantGrid` and a list of animals, both of which are returned after updated. The first input is the positive activation function used by `allMove`. The rest of the inputs to the function are used solely by `allMateHebbTime` for mating.

Inputs:

- a positive activation function
- a `PlantGrid`
- list of `Animal`'s
- list of as yet unused ID numbers
- list of determining factors for mutation
- list of random positions in the traits chromosome
- list of random positions in the weights and time delays chromosomes
- list of random `Float` type mutations
- list of random `Int` type mutations

```
>masterHebbTimeAction ::  (Float → Float) → PlantGrid → [Animal] → [Integer]
>    → [Float] → [Int] → [Int] → [Float] → [Int] → (PlantGrid, [Animal])
>masterHebbTimeAction a g fs is cs pTs pNs nFs nIs
> = (grid, energyCap (allEatAnimals (freeP,actedP++actedM)))
> where
>    (freeM,actedM) = allMateHebbTime
>       is (allDie ((deadAnimals fs)++(allMove a g (metabolism li)))) cs pTs pNs nFs nIs
>    li = livingAnimals fs
>    (grid,(freeP,actedP)) = allEatPlants g freeM
```

2.7 World: Part 2

2.7.1 Overview

Having finally defined the most important aspect of the simulation, namely the animals that inhabit it, we can now move on to tie up the remaining loose ends concerning the definition of the world in which the simulation takes place.

This is first done in the *Bins* section. The world is partitioned into bins mainly for the sake of performance, but modeling the world in this way has an effect on the properties of the environment itself. These effects are explained in the *Bins* section. Several functions for controlling the environment are also defined in this section. It is by calling functions defined in this section that the simulation is controlled.

The main program that does this is described in the *Simulation* section. It primarily uses functions from *Bins* to control the simulation, though it also accesses code from some of the other sections. Another important matter that the *Simulation* section deals with is the generation of data files. As the simulation runs, files are written containing detailed information about the state of the world at each time step. How this data is analyzed and what results are taken from it is explained in the *Results* chapter.

2.7.2 Bins

A vital aspect of the simulation is the various ways in which the animals interact. The computation involved in modeling these interactions is immense, especially when the population is large. Animals can only interact with each other within a limited range, but for an animal to know its position relative to other animals it must calculate the distance between itself and the other animal for each animal in the environment. This operation runs in quadratic time with respect to the number of animals. However, the number of animals that need to be checked is reduced (on average) by the use of `Bin`'s.

A `Bin` is a sub-environment of the main environment. In fact, the environment is partitioned into a grid of `Bin`'s of equal size. Animals can move between `Bin`'s but their senses are limited to the `Bin`'s they are currently in. This means that animals are only capable of interacting with animals in the same `Bin`, which reduces the number of distance checks that need to be performed, except in the case where all animals in the environment occupy the same Bin at the same time. An interesting consequence of this computation reducing measure is that an animal's range of senses decreases as it comes nearer to the edge of a Bin. It is only aware of the Bin it currently occupies, and completely unaware of adjacent Bin's.

The following module defines the structure of Bin's as well as a collection of Bin's. Several operations on Bin's and BinCollection's are also defined.

```
>module Bins
> (arrayWidth, arrayHeight,
> numberOfBins,
> Bin, members, plants,
> BinCollection, bins,
> niceShowBin, niceShowBinCollection,
> populateHebbTimeWorld,
> stepHebbTimeBinOrg, sortAnimals,
> fullBinOrgStep, stepBinPlant,
> extinct, getPopulation, getDeadPopulation) where
```

Bin's are high level data structures containing the entirety of the simulation. The collection of Bin's represents the environment, and the Bin's themselves contain the plants and animals that are the focus of the simulation. That is why code from the *Environment*, *Animals* and *Plants* sections is included. The *Constants* module has constants common to several modules and the *Standard Code Extensions* module has some basic data manipulation functions.

```
>import Environment -- Environment
>import Animal -- Animals
>import Plants -- Plants
>import Constants -- Constants
>import StandardExts -- Standard Code Extensions
```

Every Bin in the simulation maintains its own collection of animals and plants. The animals are contained within a list and the plants are stored within a PlantGrid. The size of the PlantGrid is derived from the size of the Bin. Each cell in a PlantGrid corresponds to a 10 by 10 unit square in the environment. The width and height for a PlantGrid are derived by dividing the width and height of the Bin by 10. The results are stored in arrayWidth and arrayHeight.

```
>arrayWidth ::  Int
>arrayWidth = (binWidth div 10)

>arrayHeight ::  Int
>arrayHeight = (binHeight div 10)
```

The number of Bin's that make up the world is derived by multiplying the width and height of the BinGrid.

```
>numberOfBins ::  Int
>numberOfBins = binGridSizeX × binGridSizeY
```

As mentioned earlier, a Bin is a collection of animals and plants. These are stored within the Bin data structure as a list of type Animal and an instance of a PlantGrid. The only Bin constructor is for an Isolated type, since each Bin is isolated from all others. Animals cannot sense other Bin's, nor the plants and animals they contain. A default Show instance is derived, though the resulting output is so messy as to be of little use. A default equality instance is also derived.

```
>data Bin = Isolated {members::[Animal], plants::PlantGrid}
>    deriving (Show, Eq)
```

The niceShowBin function provides of view of the Bin contents in a more understandable format. It outputs a string containing data about each animal (produced by niceShowAnimal from *Animals*) as well as the PlantGrid (produced by niceShowBRAMatrix). Information about each animal is printed on its own line, and then there is a blank line followed by a text representation of the PlantGrid.

Inputs:

- a Bin to be displayed

```
>niceShowBin ::  Bin → String
>niceShowBin (Isolated {members = m, plants = p})
> = (merge (map niceShowAnimal m))++''\n''++(niceShowBRAMatrix p)++''\n''
>    where
>       merge ::  [String] → String
>       merge [] = ''''
>       merge (s:ss) = s++(merge ss)
```

Individual Bin's joined together make up the entire environment. The structure that holds the individual Bin's together is a BinGrid, which is a type synonym for a BRAMatrix of Bin's. The relative positions of Bin's in the BinGrid correspond to the relative positions of Bin's in the environment. This means that when an animal wanders off the edge of one Bin, it enters the Bin whose position in the BinGrid is adjacent to the Bin along the edge being traversed.

A single BinGrid represents the entire state of the simulation at a single time step. The data stored in a BinGrid is essentially a frozen slice of time containing the entirety of the simulation.

```
>type BinGrid = BRAMatrix Bin
```

The BinGrid also has a function for displaying its contents in a tidy manner. The niceShowBinGrid function repeatedly calls the niceShowBin function and concatenates the separate strings into one large string.

Inputs:

- a BinGrid to be displayed

```
>niceShowBinGrid ::  BinGrid → String
>niceShowBinGrid b = nsbg (assocs2D b)
> where
>    nsbg ::[((Int,Int),Bin)] → String
>    nsbg [] = ''\n''
>    nsbg ((p,b):bs)
>       = (show p)++''\n''++(niceShowBin b)++''\n''++(nsbg bs)
```

The BinGrid type synonym is further encapsulated within the BinCollection data type. The purpose for creating the BinCollection data type is to allow for future expandability. Currently the only constructor for BinCollection's is the Disjoint constructor, whose only data member is a BinGrid. The BinCollection data type is therefore superfluous, but should future versions of the simulation demand for more complex types of BinCollection's with multiple properties, then the existence of a BinCollection data type will make implementation of these features easier, since a new constructor can be added to the pre-existing data type.

```
>data BinCollection = Disjoint {bins::BinGrid}
>    deriving (Show, Eq)
```

The tidy display function for `BinCollection`'s calls `niceShowBinGrid` on the `BinGrid` data member of the `BinCollection`.

Inputs:

- a `BinCollection` to be displayed

```
>niceShowBinCollection ::  BinCollection → String
>niceShowBinCollection (Disjoint b) = niceShowBinGrid b
```

An `emptyWorld` is a fresh world void of animals and plants. The `Disjoint` constructor is used to create a `BinCollection`, whose `BinGrid` is initialized such that every `Bin` contained therein uses the `Isolated` constructor with an empty list as its population and a new empty `PlantGrid` for its plants. Execution starts with this empty template, and then the initial population of plants and animals is added before the simulation begins.

```
>emptyWorld ::  BinCollection
>emptyWorld = Disjoint
> (initialize2D binGridSize (Isolated [] (newPlantGrid (arrayWidth,arrayHeight))))
```

The `populateHebbTimeWorld` function adds the initial population of animals to the empty world. Every animal created has identical genes with the exception of the sex gene, which is random. The initial values for the other genes come from *Constants* and are assigned by the `newHebbTimePopulation` function. Each animal also receives a unique ID number from a stream of `Integer`'s that is continually passed around during the course of the simulation. Also sent to the function are several lists of random values. These values assure that the starting positions, headings, neural network weights and neural network time delays are random. All of the animals also have an identical neural network architecture, which is yet another input to the function. The architecture is defined by a list of integers indicating the number of nodes at each successive layer of the network.

The function that is responsible for creating a population of animals is `newHebbTimePopulation`. The `populateHebbTimeWorld` function only assures that these animals are evenly spread across all `Bin`'s in the `BinGrid`. A fresh new `BinCollection`, the `emptyWorld`, is taken and updated with the result of the helper function `go`. The `go` function receives as input the list of animals created by `newHebbTimePopulation`, which fills the world. The number of animals created is divided by the number of `Bin`'s in the `BinGrid`, and the resulting number of animals is taken from those created by `newHebbTimePopulation` and placed into each `Bin`. This works best when the number of animals is divisible by the number of `Bin`'s, but if this is not the case then the excess animals are discarded. This is because the base case for `go` arises when there are no `Bin`'s left to add animals to, even if there still remain animals to be assigned.

Inputs:

- the number of animals to create
- a list of ID numbers for the animals
- a list of random x and y coordinates
- a list of random start headings
- a list of random floating point gene values (used for the neural network weights as well as for determining the sex gene of the animal)
- a list of random `Int` gene values (used for the neural network time delays)
- the list representation of the neural network's architecture

```
>populateHebbTimeWorld ::  Int → [Integer]
>    → [Float] → [Float] → [Float] → [Int] → [Int] → BinCollection
>populateHebbTimeWorld n is ps hs gFs gIs ns
> = gc {bins = massUpdate2D
>    gcGrid (go (assocs2D gcGrid) (newHebbTimePopulation n is ps hs gFs gIs ns))}
>    where
>       gc = emptyWorld
>       gcGrid = bins gc
>       orgPerB = n div numberOfBins

>       go ::  [((Int,Int),Bin)] → [Animal] → [((Int,Int),Bin)]
>       go [] _ = []
>       go ((p,b):bs) os = (p,b {members = x}):(go bs y)
>          where
>             (x,y) = splitAt orgPerB os
```

Once the initial conditions have been established, the simulation can begin. The simulation operates in discrete time. Everything in the environment is updated at the same time. This means that the run of the simulation can be described as a series of snapshots of the state of the environment. Since the `BinCollection` represents the entire environment, this series of snapshots is a series of `BinCollection` instances. Between time steps the entire `BinCollection` is updated.

Each world state is derived from the previous one with the help of some random data. The function that updates from one state to the next is `stepHebbTimeBinOrg`. The function takes a `BinCollection` as input and returns an updated `BinCollection`. Each `Bin` in the collection's `BinGrid` is updated independently of the others.

Since the `BinGrid` is a type of `BRAMatrix`, the update is performed with the `massUpdate2D` function. The data with which to perform the updates is generated by the `go` helper function. Each recursive step of the `go` function updates one of the `Bin`'s in the `BinGrid`. This is done by sending the `Bin`'s animal population and `PlantGrid` off to the `masterHebbTimeAction` function. The other important inputs to these functions are several lists of data, most of which are randomly generated. Each call to `masterHebbTimeAction` requires a portion of this data, and the remainder is needed for the other calls.

The first list of data holds ID numbers for new animals that are born on the given time step. Given the way that mating is handled, the maximum number of new animals that can be created is equal to the current number of animals, therefore the list of ID numbers is split at this point with the first part of the list being sent to `masterHebbTimeAction` and the remainder being used in the recursive call of `go`. All of the lists are split in this fashion, although at different points. The size of the population for the current `Bin` is assigned to `pop`. The next list contains random values used to determine whether or not mutation occurs during mating. Each chromosome of each new animal has a chance of being mutated, and since each animal has three chromosomes the number of values needed from this list for one `Bin` is 3 × `pop`. The next list also has random values, and contains valid indices within the traits chromosome. For every two animals born, the number of values needed from this list is three, because one is needed to determine the crossover point, and the other two are mutation points for the two offspring. Therefore 2 × `pop` number of values from this list is more than enough (1.5 would do, but complicates matters with type conflicts between `Float` and `Int`). The next list contains random indices for the two neural network chromosomes: weights and time delays. Like the last list, three values from this list are needed for the crossover and mutation of one chromosome type, but since the values from this list are needed by two types of chromosomes, twice as many values are needed. Multiplying 2 by 1.5 returns 3, an integer. Since this causes no type conflict problems, the number of values taken from this list per `Bin` is 3 × `pop`. The last two lists are mutation values. The first list holds mutation values of the `Float` type, which are used for mutations of both the traits chromosome and the weights chromosome. Therefore 2 × `pop` number of values are needed from this list per `Bin`. Only `pop` number of values are needed from the other list, which contains `Int` type mutations that are only used on the time delay chromosomes.

The only other input to this function is a function from `Float` to `Float`. This is the activation function for the neural networks.

Inputs:

- the `BinCollection`
- the neural network activation function
- list of ID numbers
- list of determining factors for mutation
- list of random positions in the traits chromosome
- list of random positions in the two neural network chromosomes
- list of random mutation values of `Float` type
- list of random mutation values of `Int` type

```
>stepHebbTimeBinOrg ::  BinCollection → (Float → Float) → [Integer] → [Float]
>    → [Int] → [Int] → [Float] → [Int] → BinCollection
>stepHebbTimeBinOrg bc a is cs $p_Ts$ $p_Ns$ $n_Fs$ $n_Is$
> = bc {bins = massUpdate2D bcGrid (go (assocs2D bcGrid) is cs $p_Ts$ $p_Ns$ $n_Fs$ $n_Is$)}
> where
>    bcGrid = bins bc

>    go ::  [((Int,Int),Bin)] → [Integer] → [Float] → [Int]
>       → [Int] → [Float] → [Int] → [((Int,Int),Bin)]
>    go [] _ _ _ _ _ _ = []
>    go ((pos,b):bs) is cs $p_Ts$ $p_Ns$ mFs mIs
>       = (pos,b {members = nos, plants = np}):(go bs $i_2$ $c_2$ $p_{T_2}$ $p_{N_2}$ $m_{F_2}$ $m_{I_2}$)
>       where
>          os = members b
>          p = plants b

>          pop = length os
>          ($i_1$, $i_2$) = splitAt pop is
>          ($c_1$, $c_2$) = splitAt (3 × pop) cs
>          ($p_{T_1}$, $p_{T_2}$) = splitAt (2 × pop) $p_Ts$
>          ($p_{N_1}$, $p_{N_2}$) = splitAt (3 × pop) $p_Ns$
>          ($m_{F_1}$, $m_{F_2}$) = splitAt (2 × pop) mFs
>          ($m_{I_1}$, $m_{I_2}$) = splitAt pop mIs

>          (np,nos) = masterHebbTimeAction a p os $i_1$ $c_1$ $p_{T_1}$ $p_{N_1}$ $m_{F_1}$ $m_{I_1}$
```

Animals are fully capable of moving from one `Bin` to another. When they wander outside the boundaries of one `Bin` they emerge at the proper location within the proper adjacent `Bin`. This is however not accomplished by the function `stepHebbTimeBinOrg`. Rather, this tidying operation is performed by the `sortAnimals` function, which must be called separately after each call of `stepHebbTimeBinOrg`. At the end of the `stepHebbTimeBinOrg` function's execution, it is possible for a `Bin` to contain an `Animal` whose position is outside the boundaries of that `Bin`. The `sortAnimals` function searches for such animals and reassigns each to its appropriate `Bin`.

Though simple in purpose, `sortAnimals` is a fairly complex function composed of many helper functions. This is because every time an animal moves from one `Bin` to another, the `Animal` instance must be both removed from the previous `Bin` and placed in the correct new `Bin`. This is further complicated, because an arbitrary number of animals can be removed from or added to any given `Bin`.

To deal with this issue, the `accum2D` function is used first with the `takeMember` helper function to remove all out of bounds animals from their respective `Bin`'s, and then second with the `addMember` helper function to add all of the removed animals to the proper `Bin`'s. The `addMember` function adds a single animal to a `Bin`, translating the animal's position to a place within the new `Bin` using `wrapCoordinates`, and the `takeMember` function discards a single animal from a `Bin` using the `remove` function from *Standard Code Extensions*.

The list of animals to be removed from their `Bin`'s and the list of animals to add to new `Bin`'s are generated by the `siv` helper function. The `siv` function takes a list of index/`Bin` associations and an empty accumulator as input. The accumulator is a tuple of two lists. The first list holds animals combined with the index of the `Bin` they are to be added to. For every entry in the first list there is an entry in the second list for the same animal, but with a different `Bin` index, namely the `Bin` from which the animal is to be removed from.

Each recursive call to `siv` handles one of the `Bin`'s in the `BinGrid`. The animals in each `Bin` are handled by a helper function of `siv` called `shift`. The `shift` function checks each animal with the `switchGrid` function, which returns the index of the `Bin` that an animal should be in given its current `Bin` and coordinate position. Animals that do not need to switch `Bin`'s are ignored, whereas those that do switch result in entries to both lists of the accumulator (`shift` has an accumulator that works the same as the accumulator for `siv`).

The `switchGrid` function works primarily with the help of the `checkCoordBounds` function, which tells which `Bin` a point belongs in relative to the `Bin` it is currently in. It returns one of 9 possible values, each corresponding to a `Bin` that the point could be assigned to (8 neighboor `Bin`'s plus the original `Bin`). The `place` helper function has a case for each of these values, and modifies the `Bin` coordinates accordingly. The `wrapCoordinates` function is also used, in case the original `Bin` was on an edge of the `BinGrid`, and the point crossed over this edge.

Inputs:

- a `BinCollection`

```
>sortAnimals :: BinCollection → BinCollection
>sortAnimals bc
> = bc {bins = (accum2D addMember (accum2D takeMember bcGrid removed) added)}
> where
>    bcGrid = bins bc
>    edges = matrixSize bcGrid

>    (added,removed) = siv (assocs2D bcGrid) ([],[])

>    switchGrid :: (Int,Int) → (Float,Float) → (Int,Int)
>    switchGrid (x_I,y_I) pos = place newBin (x_I,y_I)
>       where
>          newBin = checkCoordBounds (binWidth, binHeight) pos

>          place :: Int → (Int,Int) → (Int,Int)
>          place 0 (x,y) = (x,y)
>          place 1 (x,y) = wrapCoordinates edges (x - 1, y - 1)
>          place 2 (x,y) = wrapCoordinates edges (x - 1, y)
>          place 3 (x,y) = wrapCoordinates edges (x - 1, y + 1)
>          place 4 (x,y) = wrapCoordinates edges (x, y + 1)
>          place 5 (x,y) = wrapCoordinates edges (x + 1, y + 1)
>          place 6 (x,y) = wrapCoordinates edges (x + 1, y)
>          place 7 (x,y) = wrapCoordinates edges (x + 1, y - 1)
>          place 8 (x,y) = wrapCoordinates edges (x, y - 1)

>    addMember :: Bin → Animal → Bin
>    addMember b o =
>       b{members =
>          (o{position = wrapCoordinates (binWidth, binHeight) (position o)}):(members b)}

>    takeMember :: Bin → Animal → Bin
>    takeMember b o = b {members = fst (remove o (members b))}

>    siv ::  [((Int,Int),Bin)] → ([((Int,Int), Animal)], [((Int,Int), Animal)])
>       → ([((Int,Int), Animal)], [((Int,Int), Animal)])
>    siv [] changes = changes
>    siv ((pos,b):bs) (a,r) = siv bs (join++a, left++r)
>       where
>          os = members b
>          (join, left) = shift os ([],[])

>          shift ::  [Animal] → ([((Int,Int), Animal)], [((Int,Int), Animal)])
>             → ([((Int,Int), Animal)], [((Int,Int), Animal)])
>          shift [] changes = changes
>          shift (o:os) (a,r)
>             | newB ≡ pos = shift os (a,r)
>             | otherwise = shift os ((newB,o):a,(pos,o):r)
>               where
>                  newB = switchGrid pos (position o)
```

The `fullBinOrgStep` function combines the `stepHebbTimeBinOrg` and `sortAnimals` functions into a single command. This one command performs all the work of the simulation for a single time step, but only for a normal time step. That means a time step in which all the animals act as opposed to the less frequent plant growth steps (see below). The function's inputs are the same as those required by `stepHebbTimeBinOrg`.

Inputs:

- `BinCollection`
- neural network activation function
- list of ID numbers
- list of determining factors for mutation
- list of random positions in the traits chromosome
- list of random positions in the two neural network chromosomes
- list of random mutation values of `Float` type
- list of random mutation values of `Int` type

```
>fullBinOrgStep ::  BinCollection → (Float → Float) → [Integer] → [Float] → [Int]
>     → [Int] → [Float] → [Int] → BinCollection
>fullBinOrgStep bc a is cs pTs pNs nFs nIs
> = sortAnimals (stepHebbTimeBinOrg bc a is cs pTs pNs nFs nIs)
```

The `fullBinOrgStep` function executes the actions of a regular time step, in which all the animals act. Plants can be eaten during these time steps, but they only grow during plant growth steps, which occur every `growthStep` number of iterations. On a plant growth step, the animals do not act. From their points of view, new plants suddenly appear between time steps. The plant growth time step has no effect on the animals (specifically, it has no effect on their ages).

At the `BinGrid` level, the plant growth step is very similar to a regular time step. The function `stepBinPlant` makes use of a function from the *Plants* module, which is applied to each `Bin` in the `BinGrid`. The `stepBinPlant` function makes sure that the `executeGrow` function is applied to each `Bin`'s `PlantGrid`.

The `BinCollection` is passed to the function along with two lists of random data. The first is a list of random positions for plants to grow within the `PlantGrid`. The second is a list of determining factors for plant growth. The `BRAMatrix`'s `massUpdate2D` function is used to update the `BinGrid` with data generated by the go helper function. The go function returns a list of updated Bin's in tuples with their positions. The Bin's are updated by the executeGrow function.

As with stepHebbTimeBinOrg, the lists of input data are split into pieces to send off with each call to another function, in this case executeGrow. Both lists are split at the value of plantsPerGrowth, which is the maximum number of plants that can grow in each Bin on a plant growth step.

Inputs:

- BinCollection
- list of randomly generated PlantGrid positions
- list of determining factors for plant growth

```
>stepBinPlant :: BinCollection → [(Int,Int)] → [Float] → BinCollection
>stepBinPlant bc ps cs
> = bc {bins = massUpdate2D bcGrid (go (assocs2D bcGrid) ps cs)}
> where
>    bcGrid = bins bc

>    go :: [((Int,Int),Bin)] → [(Int,Int)] → [Float] → [((Int,Int),Bin)]
>    go [] _ _ = []
>    go ((pos,b):bs) ps cs
>       = (pos, b {plants = executeGrow (plants b) p1 c1}):(go bs p2 c2)
>       where
>          (p1, p2) = splitAt plantsPerGrowth ps
>          (c1, c2) = splitAt plantsPerGrowth cs
```

As long as animals are living in at least one of the Bin's there is still meaningful data being produced. However, if all animals die then there is no longer any point in continuing the simulation. When all animals are dead, they have become extinct. The extinct function below is a predicate that tests whether the population has become extinct or not. This test is performed after every time step. As long as the population is not extinct, the simulation continues. Otherwise it stops.

Since the function is executed on every time step, but has no effect on the simulation, it is important for it to execute quickly. It takes advantage of Haskell's lazy evaluation, which in this case is essentially short circuit evaluation. The function returns True exactly when the living animal list of each Bin is empty. The Boolean "and" operation is used between these checks. This means that as soon as any of the checks return a False, that is, as soon as any animal is found in any of the Bin's, no further checks need to be performed. A single False in a series of "and" comparisons makes the whole result False.

Inputs:

- a BinCollection

```
>extinct :: BinCollection → Bool
>extinct (Disjoint {bins = m}) = check bs
> where
>    bs = elems2D m

>    check :: [Bin] → Bool
>    check [] = True
>    check (b:bs)
>       = ((length (livingAnimals (members b))) ≡ 0) ∧ (check bs)
```

Two simple data gathering operations are getPopulation and getDeadPopulation. They are meant to be performed during operation rather than on generated data, and they give a useful, though limited, view of what the execution of the simulation currently looks like. The two functions are essentially the same except that one counts living animals and the other counts dead animals. Both functions make use of the `census` function, but they send different functions as input. The `getPopulation` function calls `census` with the `livingAnimals` function, and the `getDeadPopulation` function calls `census` with the `deadAnimals` function.

Inputs:

- the `BinCollection`

```
>getPopulation :: BinCollection → Int
>getPopulation (Disjoint {bins = m}) = census livingAnimals (elems2D m)
```

Inputs:

- `BinCollection`

```
>getDeadPopulation :: BinCollection → Int
>getDeadPopulation (Disjoint {bins = m})
> = census deadAnimals (elems2D m)
```

The `census` function takes as input a pruning function which extracts from a list of animals those with a particular trait. It also takes a list of `Bin`'s as input. The `census` function works by counting the number of animals returned by each application of the pruning function to the population of each `Bin`. Therefore the number of animals in the population with a special feature, as defined by the pruning function, are counted one `Bin` at a time.

Inputs:

- population pruning function
- list of `Bin`'s

```
>census :: ([Animal] → [Animal]) → [Bin] → Int
>census _ [] = 0
>census f (b:bs) = (length (f (members b))) + (census f bs)
```

2.7.3 Simulation

All modules up to this point have contained an aspect of the simulation. This module is the executable module that binds all previous modules together to run the simulation. The module name is `Main` and it contains a function called `main`, which when executed runs the simulation. These names are required by the Glasgow Haskell Compiler (GHC), which is used to compile an executable program from Haskell code. Although not all of the modules created for the simulation are directly imported into this one, all of them are in fact used (imported by other modules if not by this one).

```
>module Main (main) where

>import ActFunctions -- Neural Network Activation Functions
>import Environment -- Environment
>import Genetic -- Genetic Algorithms
>import HebbTimeNet -- Neural Networks Supporting Discrete Time Delays and Hebbian Learning
>import Animal -- Animals
>import Plants -- Plants
>import RandomStream -- Streams of Random Numbers
>import Bins -- Bins
>import Constants -- Constants
```

Before the simulation can begin, the world in which it plays out must be created. The initial state of the world depends on several constants and randomly generated values. The world that these values create is a `BinCollection`, which contains several `Bin`'s, each of which has its own population of animals and its own `PlantGrid`.

The initial animals of the population are evenly distributed among the `Bin`'s in the `BinCollection`, but their positions within those `Bin`'s are random. `Bin`'s are rectangular in shape, all with the same width and height set by `binWidth` and `binHeight` respectively. These two constants have been set such that the `Bin`'s are square in shape, for the sake of simplifying world generation.

The stream `startPositions` holds random values that are between zero and the minimum of the `binWidth` and `binHeight` inclusive. Because the `Bin`'s are square shaped, these two values must be equal, but the minimum is taken in case future modifications to the simulation make this not the case. The values generated by `startPositions` serve as both x and y coordinate positions, so having all of these values be at most the smaller of the `Bin`'s width and height assures that every point is within the boundaries of the `Bin`. These values are generated by random `seed1`.

It would of course be possible to generate two separate streams, one for x coordinates and one for y coordinates, but since these values only affect the initial conditions of the population, and because using one stream greatly simplifies matters, only one stream is used. Furthermore, the `Bin`'s being square makes it so these starting positions are approximately evenly distributed within any `Bin`.

```
>startPositions ::  [Float]
>startPositions = boundRandomFloatings 0 (min binWidth binHeight) seed1
```

The headings of the initial animals in the population are also random, though generating them is less complicated than generating the initial positions. Headings are measured in radians and can take on any value in the interval [0,2π] (2π is technically out of bounds, but is quickly fixed after one time step of the simulation). Therefore the `startHeadings` stream randomly generates values in this range using `seed2`.

```
>startHeadings ::  [Float]
>startHeadings = boundRandomFloatings 0 2π seed2
```

The final set of random values needed to establish the simulation's starting conditions are the synaptic weights in the neural networks of the initial population. There is no limit on the range of values that a synaptic weight can take on, but there is a limit on the values of the initial synaptic weights in the population. This range is [-1,1].

When the absolute value of a result from this stream is taken, it is in the range [0,1]. This is the range of all trait genes. All of the starting values of the trait genes are set by constants, except for the sex trait gene. This value is randomly generated. By using the absolute value function, the `startWeights` stream can be used to generate random starting sex trait gene values as well.

The `startWeights` stream does double duty by generating synaptic weights in the range [-1,1] and by generating numbers whose absolute values are in the range [0,1], which are used as sex trait genes. This stream uses `seed3`.

```
>startWeights ::  [Float]
>startWeights = boundRandomFloatings (-1) 1 seed3
```

The starting population is meant to be simplistic. Because of this, none of the animals in the starting population have time delays for the synapses of their neural networks (actually, all synapses have time delays of zero, which is effectively the same as having no delays). Time delays serve as a form of short term memory that informs an animal about things it recently sensed. None of the animals start with this ability. The `startTimeDelays` stream is an endless series of zeros, because the animals start with no time delays.

```
>startTimeDelays ::  [Int]
>startTimeDelays = repeat 0
```

The streams defined above are used a single time to create the initial world. After that they are discarded. The function that uses these streams to set up the initial `Bin` structure and animal population is `startWorld` (initial plants are added to the world below in `main`). The `startWorld` function is essentially a simplified call to `populateHebbTimeWorld` (see *Bins*). The streams `startPositions`, `startHeadings`, `startWeights` and `startTimeDelays` are used as inputs to `populateHebbTimeWorld`. The constant `initialPopulation` tells the function how many animals to make. The `netArchitecture` is also sent as input. The only value that `startWorld` receives as input is a list of ID numbers to assign to the initial population of animals.

The ID numbers for the animals are generated by a stream, but only a finite list is sent to `startWorld` as input. The `startWorld` function receives the front of the list of ID numbers, enough for every animal in the initial population, and the rest are used throughout the simulation.

Inputs:

- finite list of unique ID Numbers

```
>startWorld :: [Integer] → BinCollection
>startWorld is
> = populateHebbTimeWorld
>    initialPopulation is startPositions startHeadings
>    startWeights startTimeDelays netArchitecture
```

The rest of the streams defined in this module are used throughout the simulation. The front ends of the streams get used in various function calls while what remains, an infinite stream of data, gets passed on recursively.

The first of these streams is called `eventChances`. This stream holds determining factors, which are used in conjunction with probabilities to decide the outcomes of random events. The values in `eventChances` are generated by `seed4` using `smallRandomFloatings`.

```
>eventChances :: [Float]
>eventChances = smallRandomFloatings seed4
```

The next stream of values needed throughout the simulation is the `traitPositions` stream. Throughout the course of the simulation, random positions in the trait chromosome are used to decide where crossover and mutation occur. The length of the traits chromosome is given by `numberOfTraits`, so zero through (`numberOfTraits` - 1) are valid positions in the traits chromosome. Values in this range are produced by `traitPositions` with the help of `boundRandomIntegrals` using `seed5`.

```
>traitPositions :: [Int]
>traitPositions = boundRandomIntegrals 0 (numberOfTraits - 1) seed5
```

During execution, random positions within the other two chromosomes are also needed for crossover and mutation. The other two chromosomes are the synaptic weights and discrete time delays chromosomes, which both have the same length. The number of entries in these chromosomes is equal to the number of synapses defined by the neural network architecture used by all animals. This number is derived with `countSynapses`. The result from this derivation is stored in `numberOfSynapses`.

```
>numberOfSynapses :: Int
>numberOfSynapses = countSynapses netArchitecture
```

Once calculated, the `numberOfSynapses` is used by the `neuralPositions` stream to calculate random positions in both the weights and time delays chromosomes, much as the `numberOfTraits` is used to calculate random positions in the traits chromosome. The random numbers are generated by `seed6`.

```
>neuralPositions :: [Int]
>neuralPositions = boundRandomIntegrals 0 (numberOfSynapses - 1) seed6
```

The next type of random stream values used throughout the simulation are mutation values. When mutation occurs, a random value is added to a random gene value of a chromosome. The traits and weights chromosomes hold `Float` values, and the time delays chromosome holds `Int` values, so two streams of random mutations are needed. `Float` mutations are made by `floatMutations` using `boundRandomFloatings` and `seed7`. `Int` mutations are made by `intMutations` using `boundRandomIntegrals` and `seed8`.

```
>floatMutations ::  [Float]
>floatMutations = boundRandomFloatings (-maxFloatMutation) maxFloatMutation seed7

>intMutations ::  [Int]
>intMutations = boundRandomIntegrals (-maxIntMutation) maxIntMutation seed8
```

The last random stream is actually two streams combined into one. The `plantPositions` stream holds random positions for plants to grow at. Each position is a 2-tuple of `Int` values. Unlike the animal starting positions generated by `startPositions`, two separate streams with different seeds, `seed9` for x-coordinates and `seed10` for y-coordinates, are used by `plantPositions`. This is done because random plant positions are used throughout the simulation, not just at the start. Two different streams must be used, because otherwise the x and y coordinates would not be probabilistically independent, which limits the range of 2-tuples that `plantPositions` would be able to generate. Two `boundRandomIntegrals` streams are zipped together to create the stream of plant position tuples.

```
>plantPositions ::  [(Int,Int)]
>plantPositions
> = zip
>    (boundRandomIntegrals 0 (arrayWidth - 1) seed9)
>    (boundRandomIntegrals 0 (arrayHeight - 1) seed10)
```

The random positions in `plantPositions` are used both in setting up the initial world and throughout the simulation. On every plant growth time step, several random positions in every `PlantGrid` are chosen as potential positions for new plants to grow at. The initial plants in the world are created by an initial series of plant growth time steps. The function that carries out a plant growth time step is `plantGrow`.

The `plantGrow` function updates a `BinCollection` using the streams `eventChances` and `plantPositions`. Once the `BinCollection` is updated, it is returned in a tuple along with the values from the two streams that were not used. Returning the streams allows new random values to be taken from them. The `BinCollection` is updated by `stepBinPlant`, to which enough values from the two streams are sent to satisfy the function's needs.

Inputs:

- the `BinCollection`
- stream of values from `eventChances`
- stream of values from `plantPositions`

```
>plantGrow ::  BinCollection → [Float] → [(Int,Int)] → (BinCollection,[Float],[(Int,Int)])
>plantGrow pg cs ps = (npg, c₂, p₂)
>    where
>       plants = plantsPerGrowth × numberOfBins
>       (c₁, c₂) = splitAt plants cs
>       (p₁, p₂) = splitAt plants ps
>       npg = stepBinPlant pg p₁ c₁
```

The function that makes the initial series of calls to `plantGrow`, which establishs the starting plant population, is `startGrow`. It is a simple recursive function with a counter that decreases by one on each call. The `plantGrow` function is called on every call to `startGrow`, and the output that it returns is used as the input on the recursive call to `startGrow`. Once the counter reaches zero, the data returned by `plantGrow` becomes the output of `startGrow`.

Inputs:

- the `BinCollection`
- stream of values from `eventChances`
- stream of values from `plantPositions`
- the number of plant growth time steps to execute

```
>startGrow ::  BinCollection → [Float] → [(Int,Int)]
>    → Int → (BinCollection,[Float],[(Int,Int)])
>startGrow pg cs ps 0 = (pg,cs,ps)
>startGrow pg cs ps n = startGrow npg c2 p2 (n - 1)
>    where
>       (npg, c2, p2) = plantGrow pg cs ps
```

The next function is the `main` function, the execution of which runs the simulation. As mentioned above, the function is named `main` to be in accordance with GHC's standard expectations. Once the program is compiled and run, it is the `main` function which is called and executed. As is required, it returns the general IO type.

The `main` function's primary purpose is to set up the initial world using the streams and functions above, and then send this data to the `loop` function, which calls itself endlessly as long as the simulation runs. The `main` function sets up the world using a `let` clause. The first line of the `let` clause splits the stream [0, ...] into two lists. The first list holds enough ID numbers to make the initial population of animals when sent as the input to `startWorld`. The `BinCollection` returned by `startWorld` is then sent to `startGrow` to go through `initialPlants` number of plant growth time steps. The updated `BinCollection` and what remains of the streams `eventChances` and `plantPositions` are returned.

Once the starting conditions for the world are set, it becomes input to the `loop` function. What remains of the `eventChances` and `plantPositions` streams, as well as the rest of the ID number stream, is sent to `loop` as well. Also sent are all other streams used throughout the simulation: `traitPositions`, `neuralPositions`, `floatMutations` and `intMutations`. The last inputs to `loop` are the number zero and the Boolean value `False`. Zero is the starting value for the simulation counter and `False` is used to control the plant growth time steps. The `loop` function calls itself as long as the simulation runs. Under ideal situations this goes on forever, so that the simulation can only be stopped by user intervention. However, if the death of all animals terminates the simulation, then the `loop` function will return to `main`, which prints "done" to the screen.

```
>main ::  IO ()
>main =
> do
>    let
>    (start,ids) = splitAt initialPopulation [0, ...]
>    world = startWorld start
>    (nw, c2, p2)= startGrow world eventChances plantPositions initialPlants

>    loop nw ids c2 traitPositions neuralPositions floatMutations intMutations p2 0 False

>    print ''done''
```

While the `main` function is the one executed to run the simulation, the work of the simulation occurs in the `loop` function. It has three cases: one for when all animals die, one for a plant growth step and one for a regular time step. Only when all animals die does the function terminate. Otherwise it continues to call itself recursively. The function knows that all animals are dead when the number of living animals is zero. To distinguish between the other two cases the function's counter and a Boolean value are used. Whenever the counter is divisible by growthTimeStep and the Boolean value is True, it is time to perform a plant growth time step. After a plant growth time step, the counter is not incremented. It only measures the number of regular time steps. Therefore the Boolean value is needed to prevent plant growth time steps from occurring repeatedly. On a plant growth time step the Boolean value is set to False, so that the next time step will be a regular time step even though the counter is still divisible by growthTimeStep. Regular time steps set the Boolean value back to True, and repeat until the counter is once again divisible by growthTimeStep.

The plant growth time step is easy to execute by calling plantGrow. The results from this function are used in the recursive call to loop, and the Boolean value is set to False. The regular time step case is slightly more complicated because the correct number of values must be taken from several streams and then sent to fullBinOrgStep as input, but this case works essentially the same in that it calls another function to perform the update of the simulation's world, and then makes a recursive call to loop with updated data, an incremented counter, and a Boolean value of True. Notice that the activation function used in the call to fullBinOrgStep is actPiecewisePos.

A regular time step makes use of the writeFile function to log data about the simulation. A new file is written for every regular time step. The name of each file is "data" followed by the counter value, which is the time step. The data printed to these files is generated by niceShowBinCollection, which displays all pertinent information from the BinCollection in a format that is easy to analyze.

Inputs:

- the BinCollection
- stream of as yet unused ID numbers
- stream of determining factors
- stream of random traits chromosome positions
- stream of random positions in the weights and time delays chromosomes
- stream of random Float type mutations
- stream of random Int type mutations
- stream of random PlantGrid positions for plants to grow at
- the number of time steps elapsed so far (the counter)
- a Boolean value indicating whether or not a plant growth time step can occur

```
>loop ::  BinCollection → [Integer] → [Float] → [Int] → [Int]
>     → [Float] → [Int] → [(Int,Int)] → Integer → Bool → IO ()
>loop bc is cs $p_Ts$ $p_Ns$ $m_Fs$ $m_Is$ ps n b
> | living ≡ 0 =
>    do
>       print ‘‘all dead’’
> | b ∧ (n mod growthTimeStep ≡ 0) =
>    do
>       let
>          (nbc, $c_2$, $p_2$) = plantGrow bc cs ps

>       print ‘‘plant cycle’’

>       loop nbc is $c_2$ $p_Ts$ $p_Ns$ $m_Fs$ $m_Is$ $p_2$ n False

> | otherwise =
>    do
>       let
>          ($i_1$, $i_2$) = splitAt pop is
>          ($c_1$, $c_2$) = splitAt (3 × pop) cs
>          ($p_{T_1}$, $p_{T_2}$) = splitAt (2 × pop) $p_Ts$
>          ($p_{N_1}$, $p_{N_2}$) = splitAt (3 × pop) $p_Ns$
>          ($m_{F_1}$, $m_{F_2}$) = splitAt (2 × pop) $m_Fs$
>          ($m_{I_1}$, $m_{I_2}$) = splitAt pop $m_Is$

>          nbc = fullBinOrgStep bc actPiecewisePos $i_1$ $c_1$ $p_{T_1}$ $p_{N_1}$ $m_{F_1}$ $m_{I_1}$

>       print ‘‘regular cycle’’
>       print n

>       writeFile (‘‘data/data’’++(show n)++‘‘.txt’’) (niceShowBinCollection nbc)

>       loop nbc $i_2$ $c_2$ $p_{T_2}$ $p_{N_2}$ $m_{F_2}$ $m_{I_2}$ ps (n+1) True

>    where
>       pop = living + (getDeadPopulation bc)
>       living = getPopulation bc
```

Chapter 3

Results

Having defined the simulation in its entirety, we now move on to the results. For each trial we first present the data it generated. Each trial uses a different set of constants, which are provided at the beginning of each data set. Because a single trial generates an enormous amount of data, we choose to only present data pertinent to our analysis. Pertinent data consists of anything which is used to interpret the data from the simulation. Should one be curious, more data from these trials is available on the included CD.

Each trial within this section represents the data generated by the simulation for a given set of starting constants. Because of how we defined our random number generators in *Streams of Random Numbers*, these results can be reproduced by use of the same constants and random seeds. Because the number of constants used by the simulation is very large, there are many possibilities which are not explored below. The trial runs chosen to appear below are those which demonstrate interesting interactions of some sort. Therefore different data is highlighted within each of the trials below in correspondence with what is interesting about that trial. Other data has been left out. What these trials do not depict are the effects of changing single constants upon the dynamics of the system. This is partly done because such an analysis for this many constants would be intractable, but also because observation implies that at least some of the constants exhibit the "Butterfly Effect", namely that even small changes to a constant can radically change the quantitative, and sometimes even the qualitative, behavior of the system (much like a storm being caused miles away as a result of a butterfly flapping its wings). Therefore the trials below should be seen as a mere sampling of some of the interesting possibilities of this simulation.

Each section presents first the data from a trial and then an interpretation of the data. The data sections consist of graphs of data along with captions pointing out interesting features of the data. Each of these sections starts with a table of the constants used to generate the results for the given trial. Please refer to the *Constants* section for information on the meaning of each constant. The interpretation sections go into more depth as to possible causes and meanings of the resulting data.

3.1 Trial 1: Volterra-Lotka Behavior

This trial ran for 75971 time steps before every animal died. The population started out exclusively as herbivores, and out of that population evolved several other eating behaviors, though the only other diet type to maintain a persistent presence was the herbivore-carnivore combination. Of interest is the periodic behavior of the population graphs. In particular, both the relationship between the population of herbivores and the population of herbivore-carnivores, and the relationship between all living animals and the plant population resemble the dynamics of Volterra-Lotka predator-prey equations, especially near the end of the simulation. Volterra-Lotka equations are explained in more detail in the interpretation of this data.

3.1.1 Data

interactionDistance = 10	minMatingTime = 0	growthTimeStep = 5
speciesEqualityConstant = 0.15	maxMatingTime = 7	plantsPerGrowth = 90
minStartEn = 0.1	sStartEnergy = 0.5	plantNutrition = 65
maxStartEn = 0.9	sSight = 0.0	binGridSizeX = 4
minSight = 15	sHerbivore = 0.6	binGridSizeY = 3
maxSight = 70	sCarnivore = 0.4	binWidth = 100
minManeuverability = 0.1	sCannibal = 0.4	binHeight = 100
maxManeuverability = 3.0	sCarrion = 0.4	initialPopulation = 150
maneuverabilityRestriction = 0.1	sManeuverability = 0.0	initialPlants = 200
minMoveSpeed = 1.0	sSpeed = 0.0	netArchitecture = [26,26,26,2]
maxMoveSpeed = 5.0	sLifespan = 0.3	maxFloatMutation = 0.25
minMove = 0.2	sMutation = 1.0	maxIntMutation = 3
moveCostMultiple = 0.05	sRadius = 0.0	initialEnergy = 200
minMaturity = 30	sMateCost = 0.2	metabolicRate = 0.057
maxMaturity = 60	sLearning = 0.1	decay = 0.3
minLifespan = 300	sMaturity = 0.5	seed1 = 17231
maxLifespan = 1000	sMateTime = 0.5	seed2 = 27372
minMutationRate = 0.1	sDummy = 0.5	seed3 = 34183
maxMutationRate = 1.0	sMateRecovery = 0.5	seed4 = 57124
minRadius = 5	deathEnergy = 0.7	seed5 = 16525
maxRadius = 20	energyRestriction = 23	seed6 = 24916
minMatingCost = 50	predationMultiplier = 19	seed7 = 27717
maxMatingCost = 300	predationConstant = 150	seed8 = 82148
minMatingRecovery = 50	carrionMultiplier = 17	seed9 = 56729
maxMatingRecovery = 100	cannibalGracePeriod = 30	seed10 = 33220

Constants for trial 1

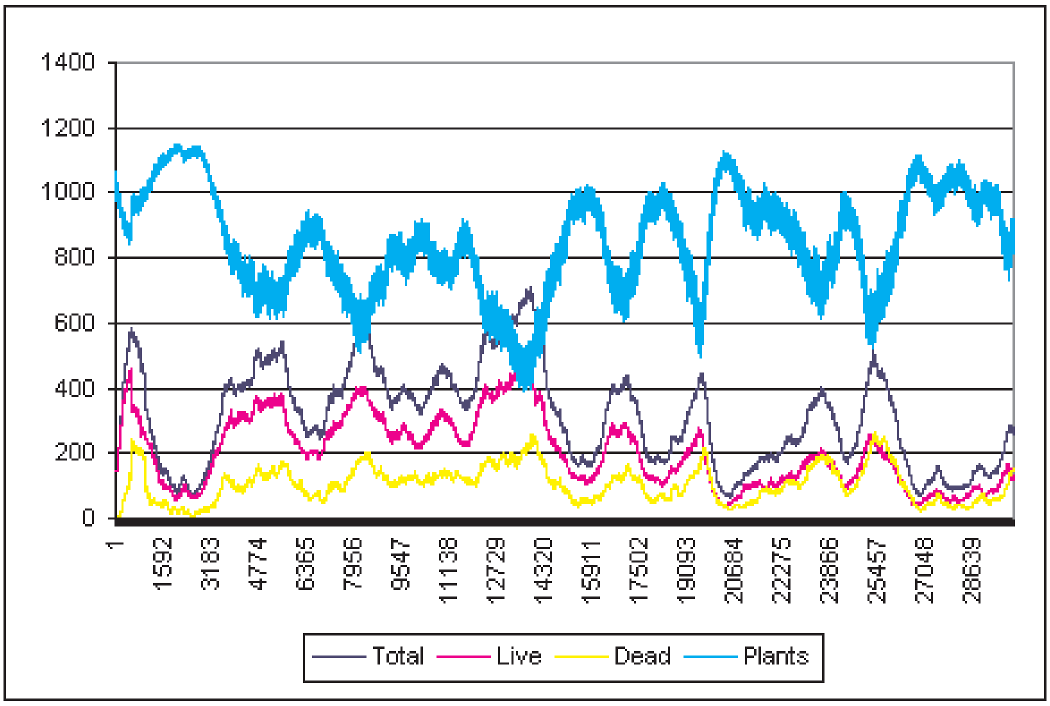


Chart of the total animals, live animals and dead animals, along with the number of plants, for time steps 0 – 30000. Notice how the plant population drops whenever the animal population is high, and is high whenever the animal population is low.

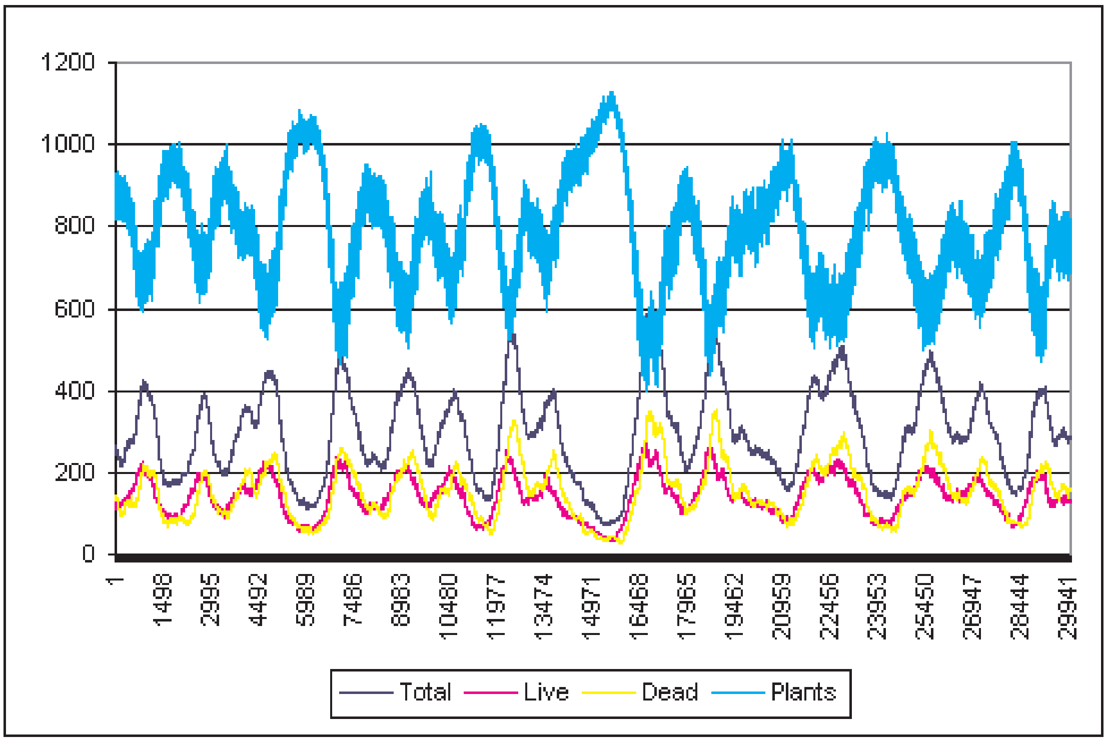


Chart of the total animals, live animals and dead animals, along with the number of plants, for time steps 30001 – 60000. The periodic cycling of the plant and animal populations becomes more stabile and regular. In other words, the time between peaks and valleys in the graphs fluctuates less.

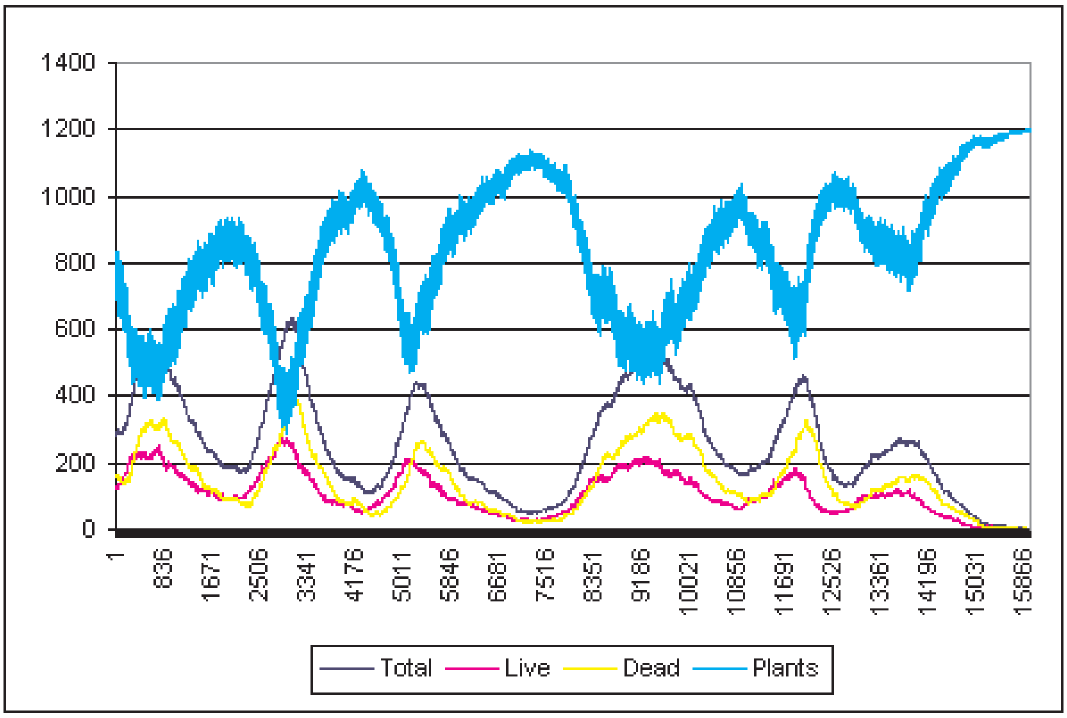


Chart of the total animals, live animals and dead animals, along with the number of plants, for time steps 60001 – 75971. The relationship between the live animal population and the plant population is now clearly seen to resemble the dynamics of the Volterra-Lotka system. This is reasonable because nearly all animals maintain the herbivore eating behavior throughout the simulation. In one of the valleys of the cycle the animal population is unable to recover, so it dies out. As this happens, the plant population approaches maximum capacity.

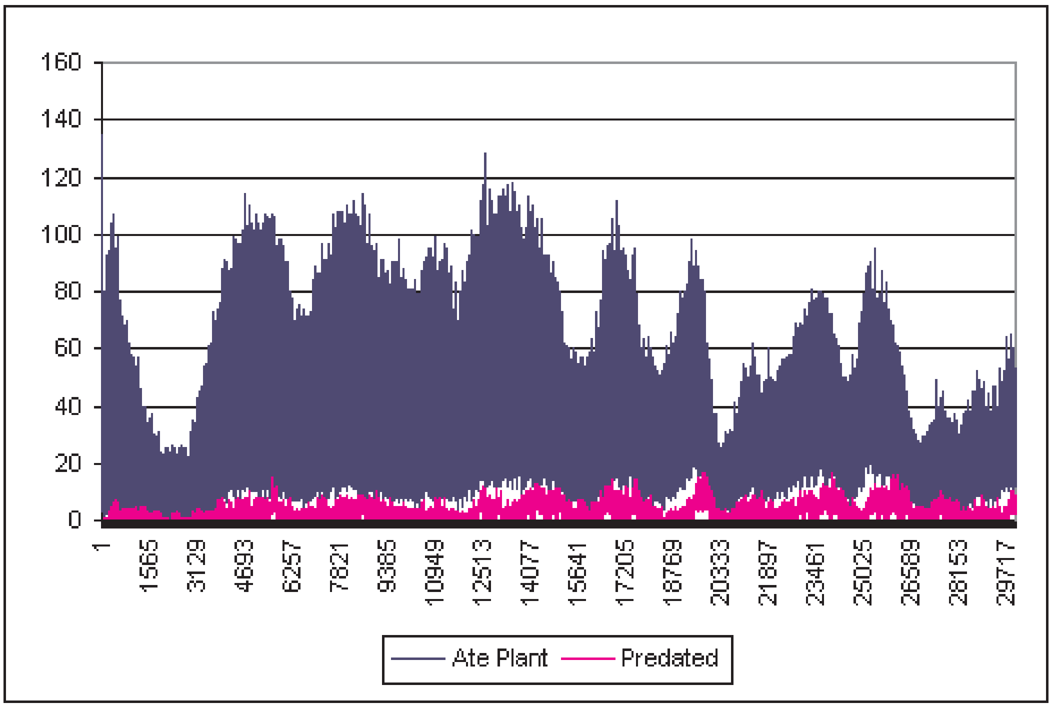


Chart of the number of animals that ate plants and the number that ate an animal of another species (Predated), for time steps 0 – 30000. It is clear that eating plants is immensely more popular than predation. The number of animals that eat plants fluctuates by large amounts in the short term because new plants grow only at the beginning of each plant growth time step. This is why the plot for "Ate Plant" is so *thick*.

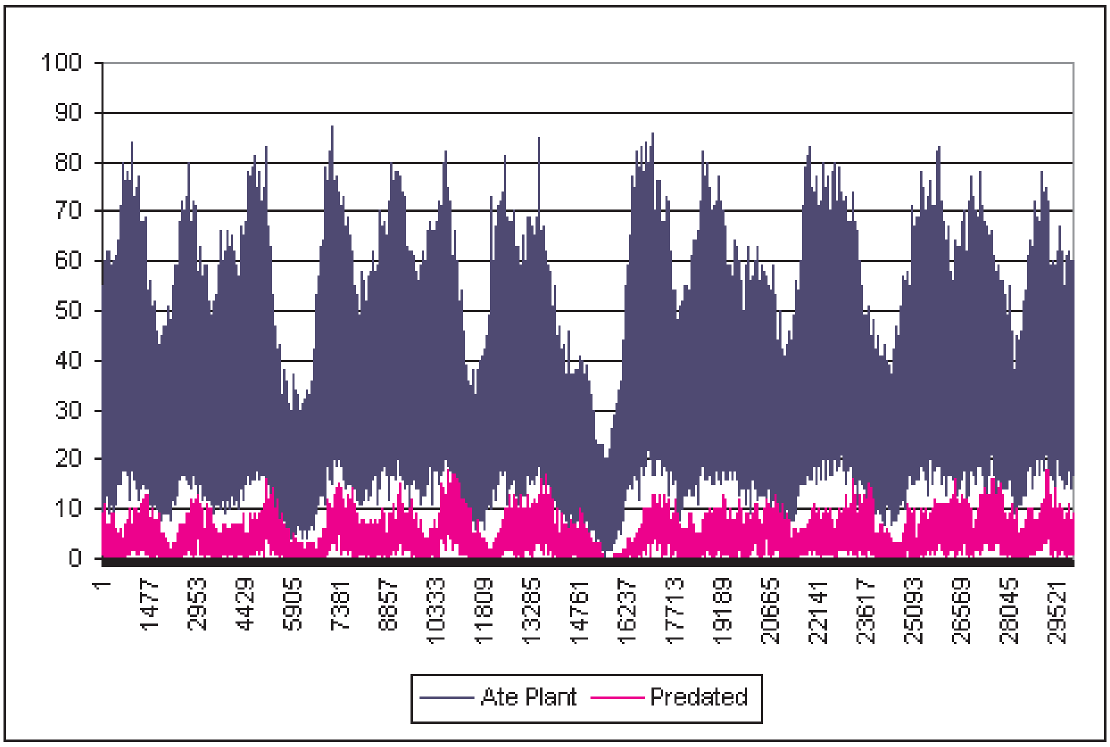


Chart of the number of animals that ate plants and the number that ate an animal of another species (Predated), for time steps 30001 – 60000. Both of these plots start to cycle more regularly, with a period that appears to be in correspondence with that of the population fluctuations for the entire population.

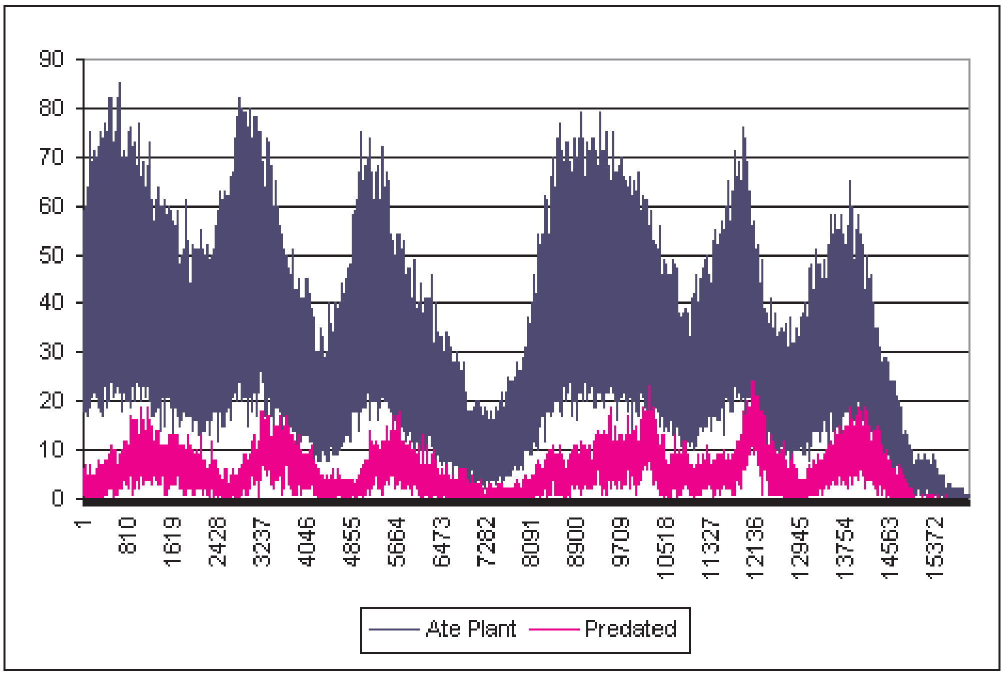


Chart of the number of animals that ate plants and the number that ate an animal of another species (Predated), for time steps 60001 – 75971. The fluctuations in the number of animals that predate remains fairly constant even as cycles for the number of animals eating plants begin having smaller peak values and deeper valleys. At one point this value sinks particularly low, but then recovers. The second time this occurs, there is no recovery. Predation cuts off slightly before plant eating, and then everything is dead. The cut off of predation could indicate a lack of prey to eat, requiring the herbivore-carnivores to depend on plants for sustenance. It could also indicate a lack of predators. The graphs below indicate that it is in fact the prey population that died out first.

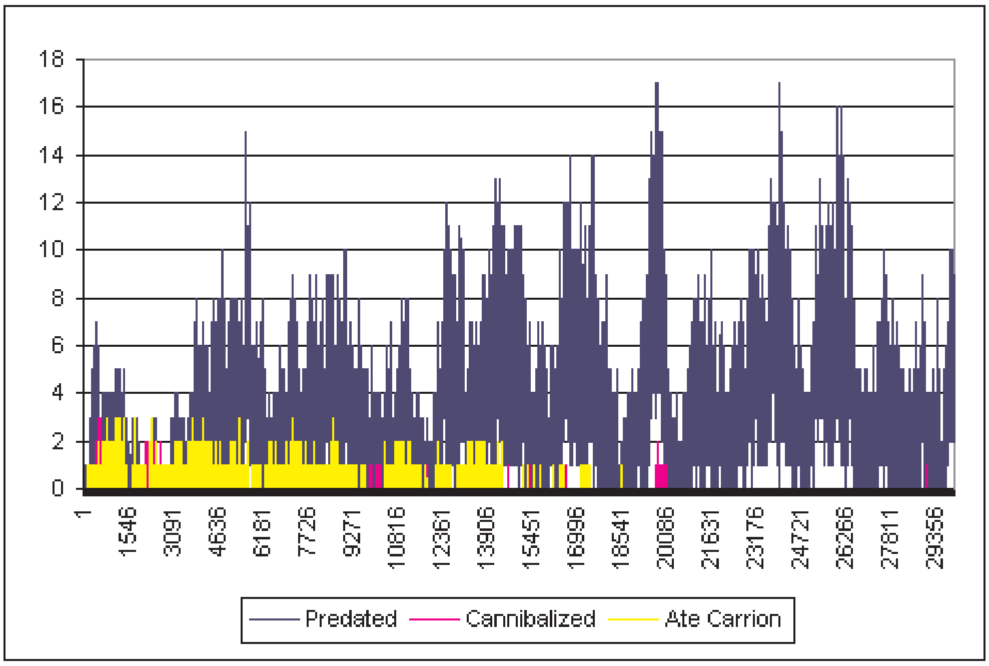


Chart of the number of animals that ate an animal of another species (Predated), an animal of the same species (Cannibalized) or a dead animal (Ate Carrion), for time steps 0 – 30000. Carrion eating becomes fairly popular at the start of the simulation, but soon disappears. Some isolated incidents of cannibalism occur, but regular predation remains more popular than both of these eating behaviors.

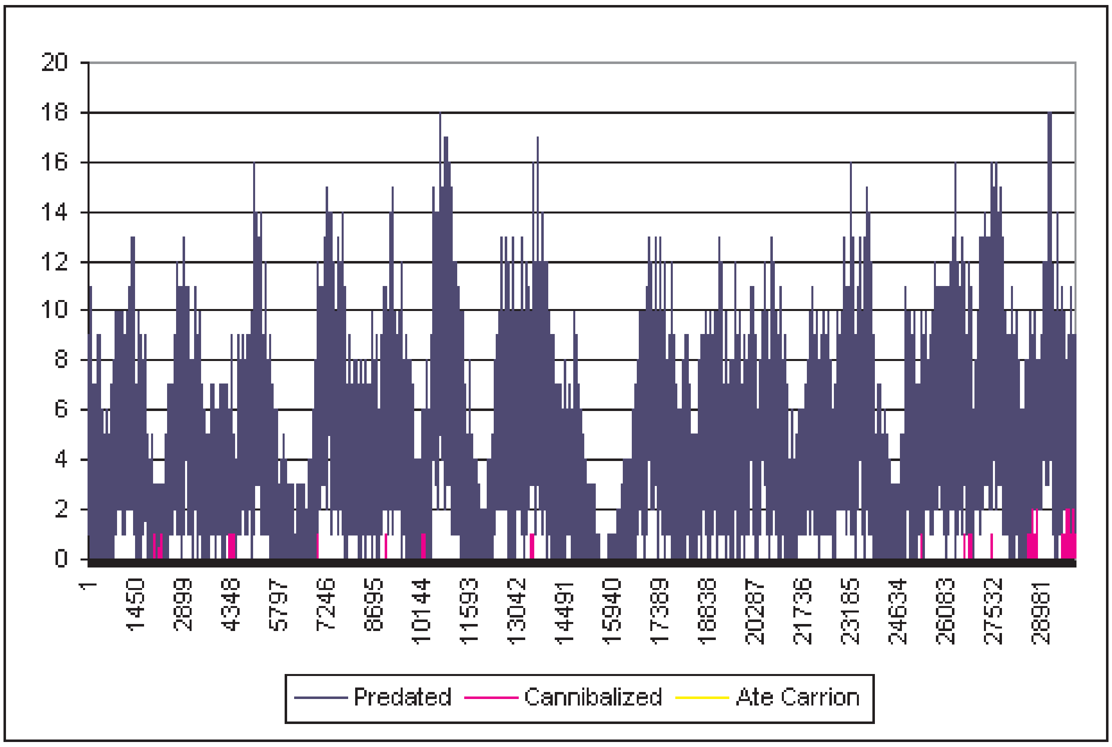


Chart of the number of animals that ate an animal of another species (Predated), an animal of the same species (Cannibalized) or a dead animal (Ate Carrion), for time steps 30001 – 60000. Carrion eating has disappeared completely, but cannibalism still occurs, although rarely.

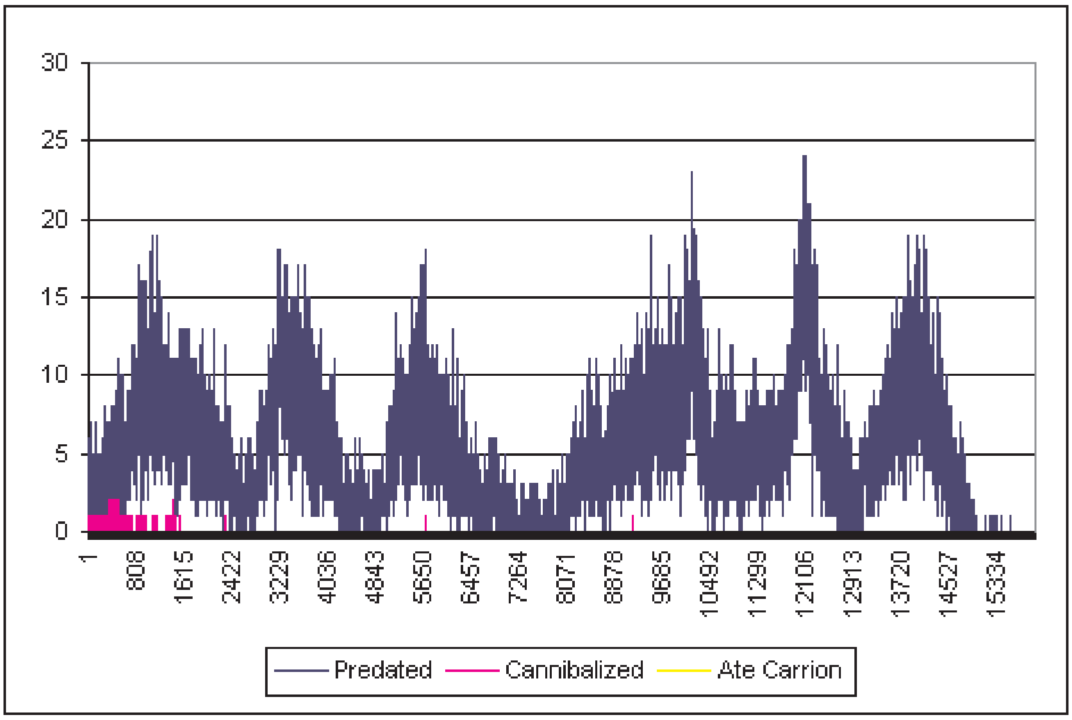


Chart of the number of animals that ate an animal of another species (Predated), an animal of the same species (Cannibalized) or a dead animal (Ate Carrion), for time steps 60001 – 75971. Carrion eating does not emerge in the population again, and cannibalism dies out almost completely, but there remain rare isolated occurrences. By the time the population dies out, cannibalism seems to have disappeared completely.

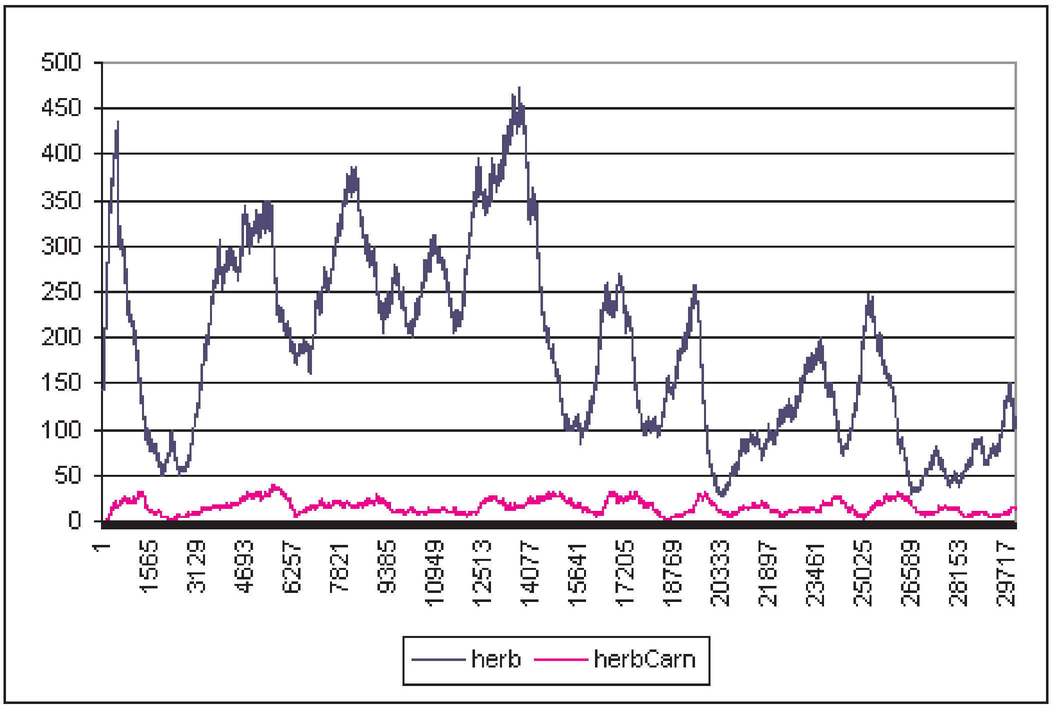


Chart of the number of pure herbivores and herbivore-carnivores, for time steps 0 – 30000. The herbivore population fluctuates wildly at some points, although periodically. The herbivore-carnivore population emerges soon after the start of the simulation and remains consistently small, although several of its peaks do appear to occur shortly after points where the herbivore population begins to decrease.

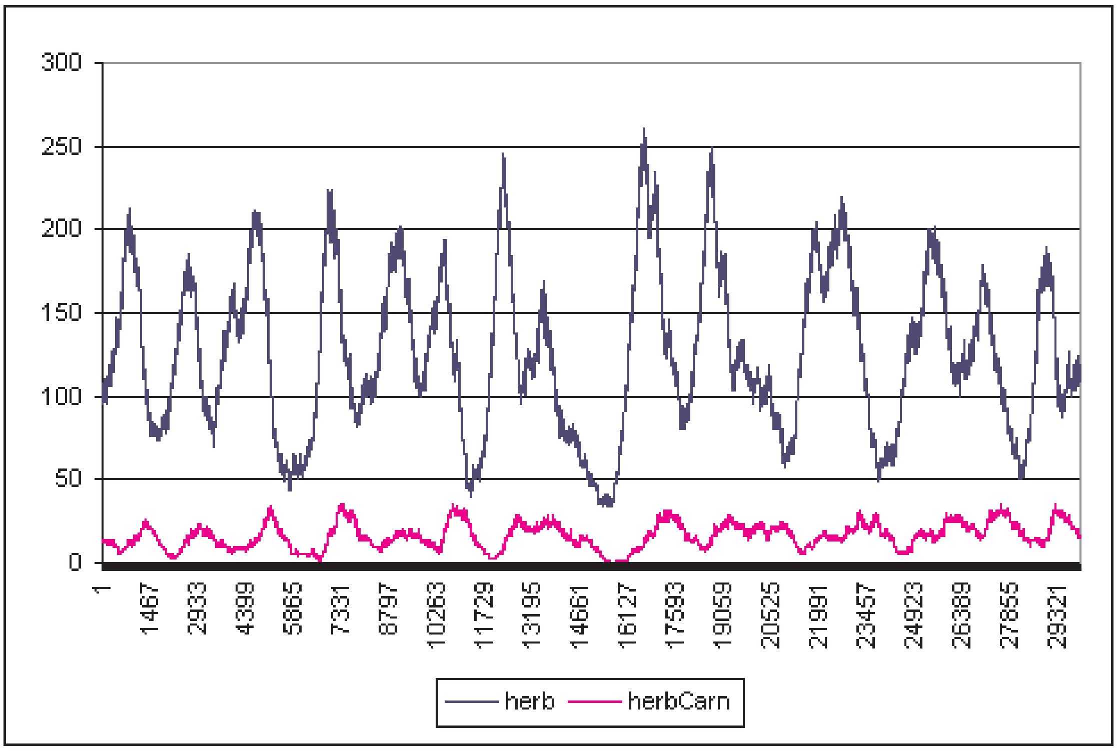


Chart of the number of pure herbivores and herbivore-carnivores, for time steps 30001 – 60000. In this timeframe each peak of the herbivore-carnivore population corresponds to a peak of the herbivore population, in that the predator population peaks soon after the prey population peaks. The valleys correspond in a similar manner. One sees the same dynamics in the Volterra-Lotka system of predator-prey differential equations.

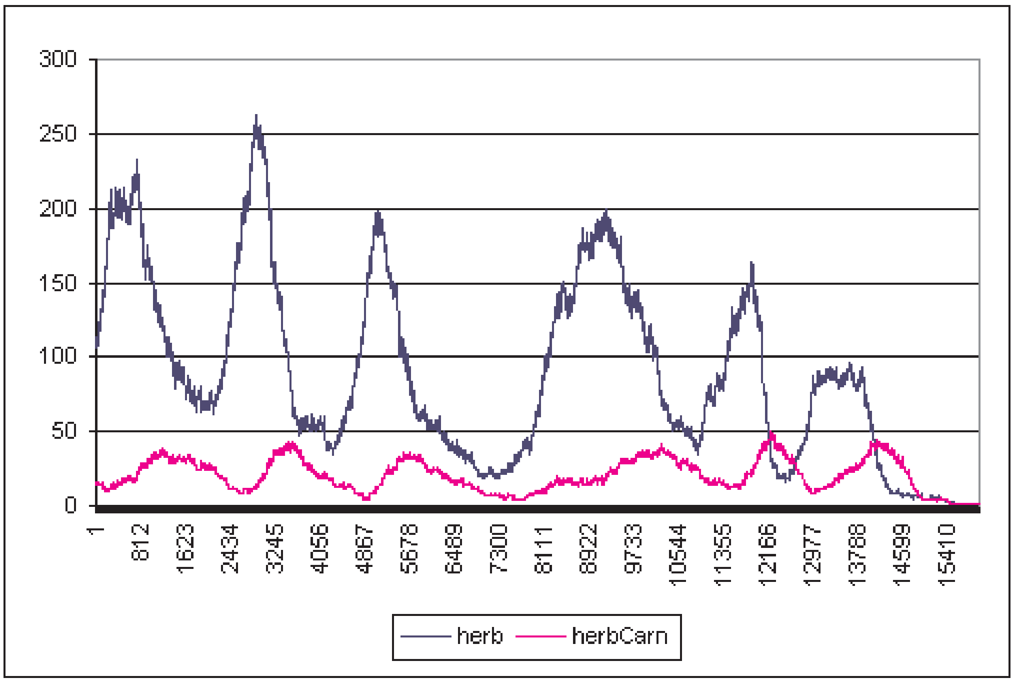


Chart of the number of pure herbivores and herbivore-carnivores, for time steps 60001 – 75971. The correspondence between the peaks and valleys of the two populations is clearer than ever. The herbivore-carnivore population starts peaking at slightly higher values than previously, whereas each consecutive peak of the herbivore population (excluding the first) is at most the height of the previous peak. Eventually the valleys of the herbivore population drop below the herbivore-carnivore population. The second time this happens, the prey population cannot recover, and dies out. The predator population soon follows suit (even though herbivore-carnivores are also able to eat plants).

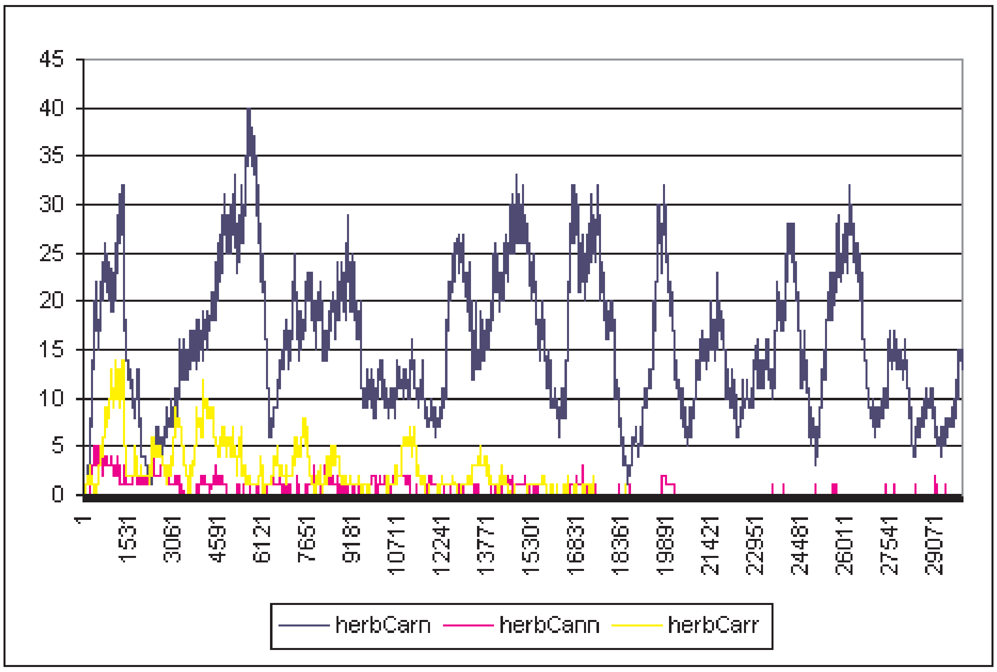


Chart of the number of herbivore-carnivores, herbivore-cannibals and herbivore-carrion eaters, for time steps 0 – 30000. Herbivore-carrion eaters appear soon after the start of the simulation, but eventually die out. Herbivore-cannibals are also present early on, and also die out, but the behavior keeps emerging again and again in the population, only to disappear again after a few time steps. This explains the intermittent instances of cannibalism observed earlier.

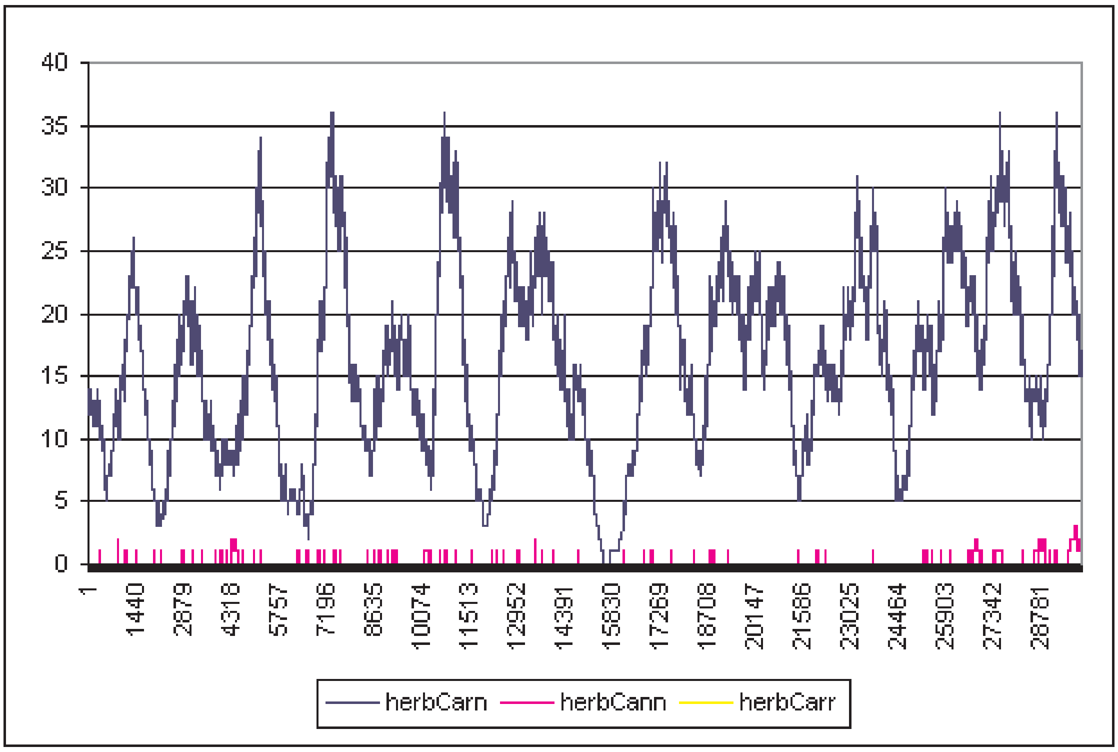


Chart of the number of herbivore-carnivores, herbivore-cannibals and herbivore-carrion eaters, for time steps 30001 – 60000. Herbivore-carrion eaters remain extinct, but herbivore-cannibals keep reappearing, often no more than one at a time.

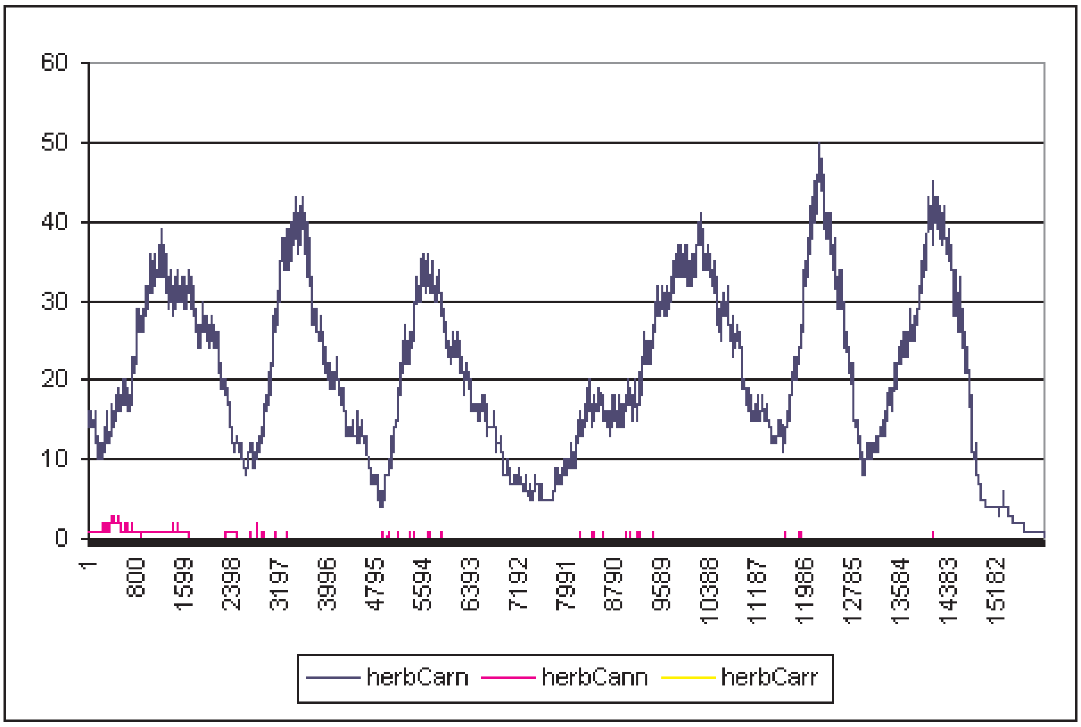


Chart of the number of herbivore-carnivores, herbivore-cannibals and herbivore-carrion eaters, for time steps 60001 – 75971. The herbivore-cannibal population enjoys a slightly longer period of existence than previously, but then returns to consisting only of rare individuals that pop up and disappear on their own from time to time. Herbivore-carrion eaters remain extinct.

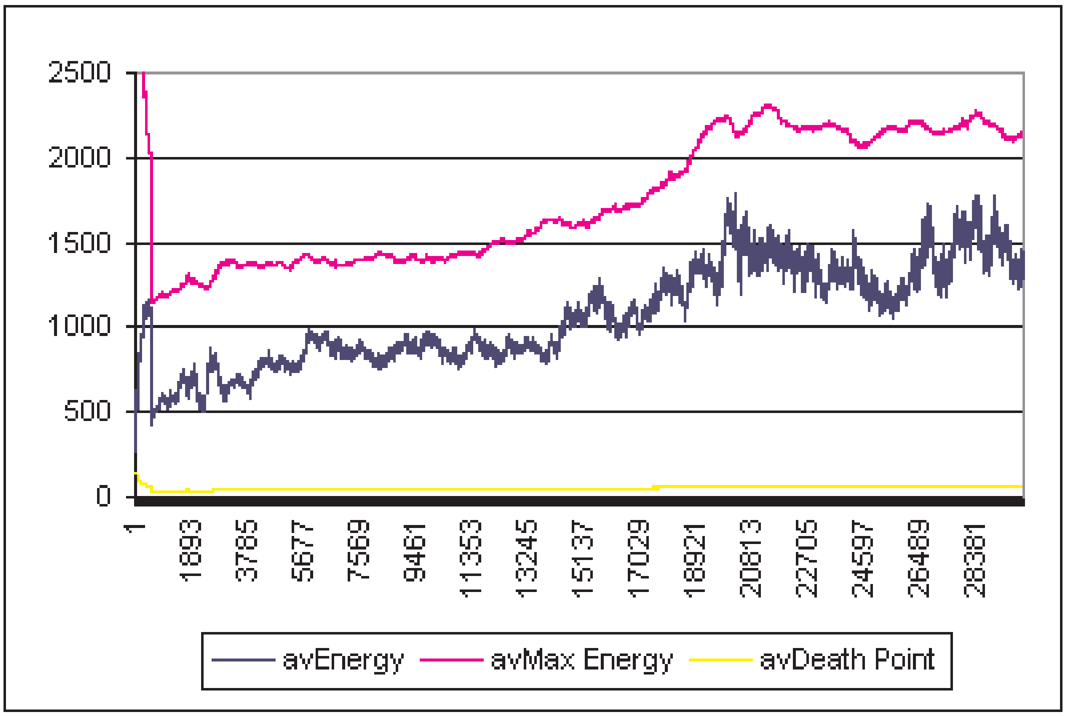


Chart of average energy levels, max energy levels and death points, for time steps 0 – 30000. The energy levels start outside the display range of the graph because the initial population is artificially assigned its initial energy. This amount happens to be quite high, and results in a large maximum energy level as well. After the first generation dies out these values are much lower. However, the average maximum energy level increases quickly, allowing the average energy level to take on larger values. The average death point also increases over time, but never becomes terribly large.

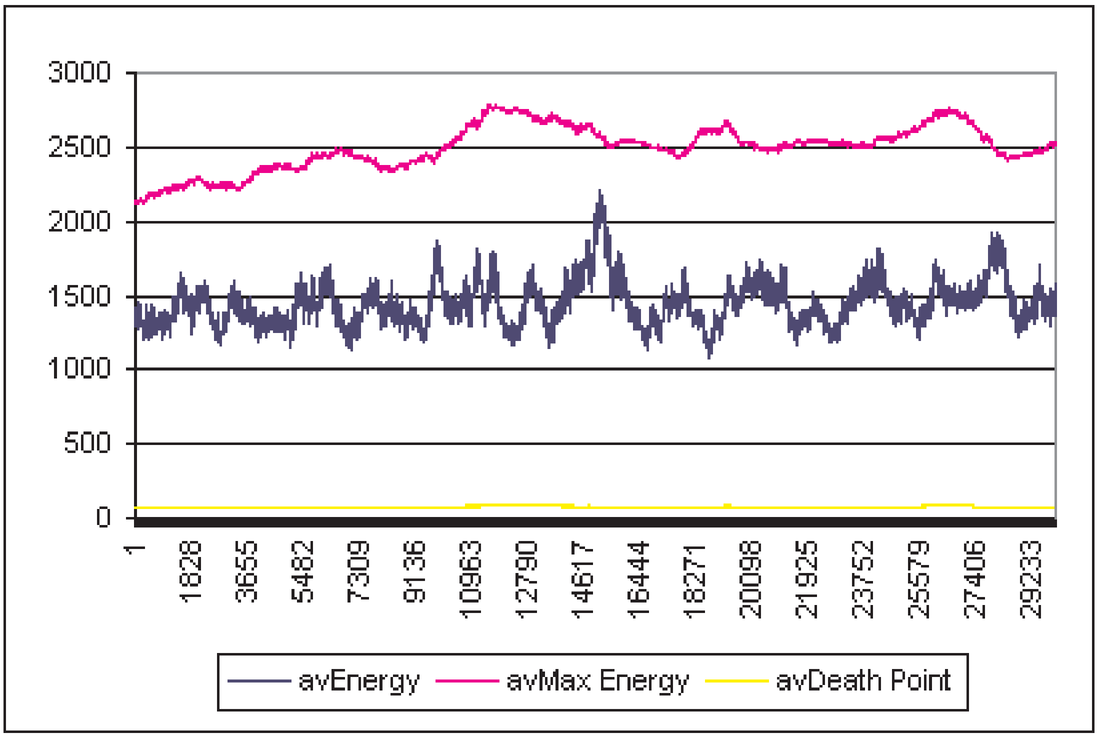


Chart of average energy levels, max energy levels and death points, for time steps 30001 – 60000. The average maximum energy level begins to even out and the average death point remains small. The average energy level fluctuates wildly between these two, but stays nearer to the maximum energy level.

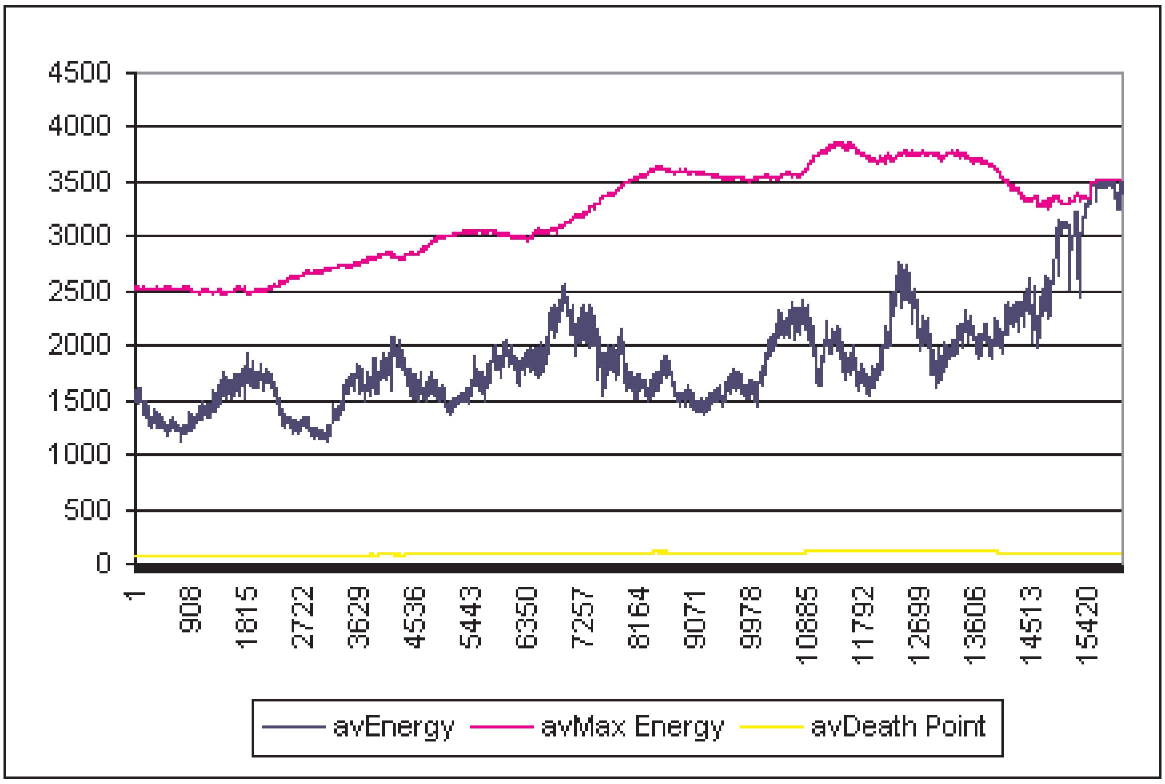


Chart of average energy levels, max energy levels and death points, for time steps 60001 – 75971. The average maximum energy level increases even more as the average death point remains small. At the end of the simulation the average energy level meets to maximum energy level, meaning that the last animals in the world could not have died of energy loss, but rather must have died from old age.

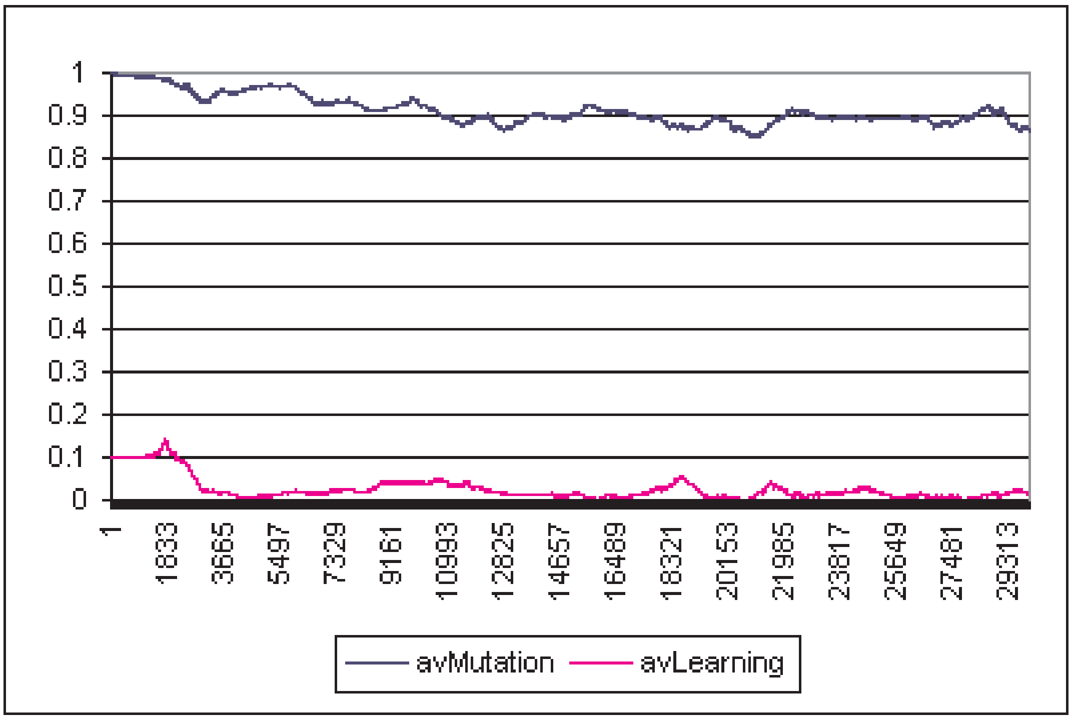


Chart of average mutation and learning rates, for time steps 0 – 30000. The average mutation rate starts at 1 and decreases slightly. The average learning rate starts at 0.1, and though already small, it decreases as well.

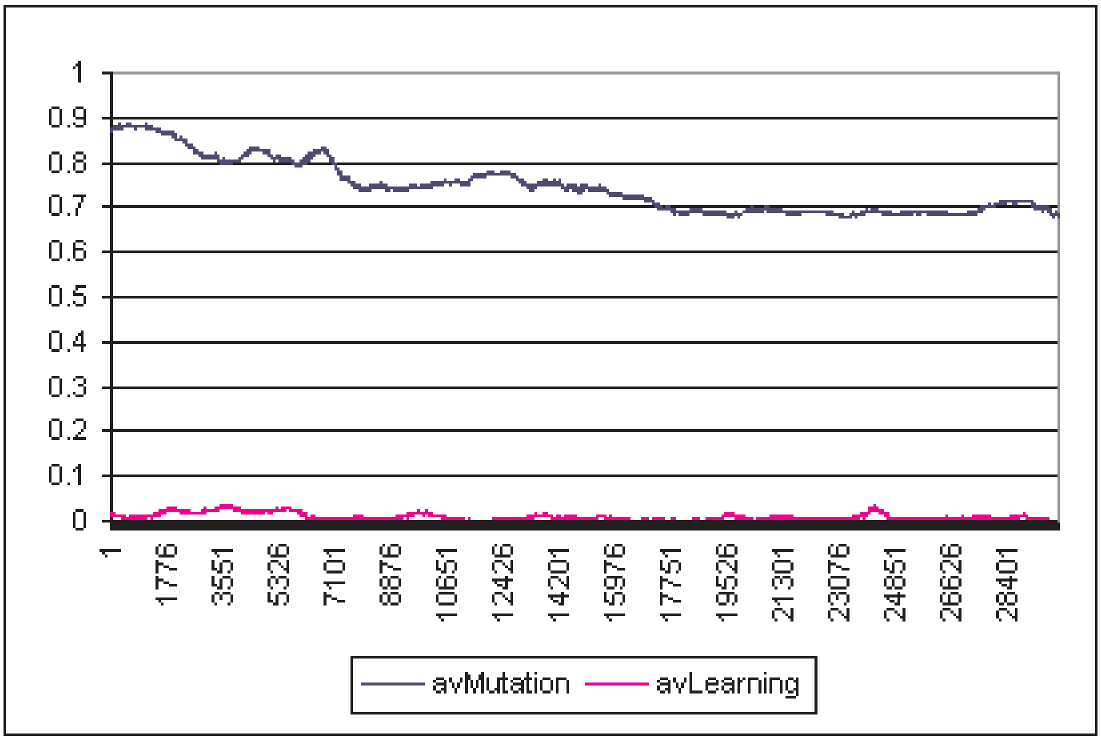


Chart of average mutation and learning rates, for time steps 30001 – 60000. Both the average learning and mutation rates continue to decrease. The average learning rate stays very close to zero and the average mutation rate becomes ever smaller, though gradually.

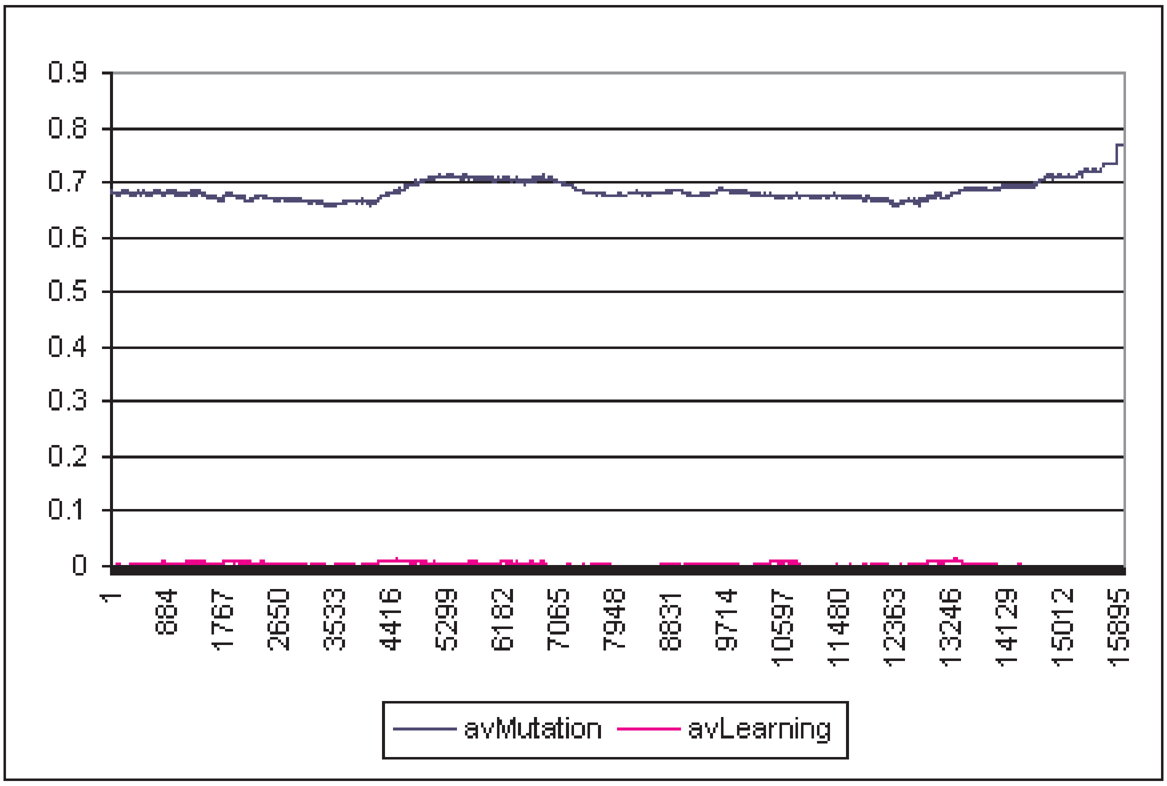


Chart of average mutation and learning rates, for time steps 60001 – 75971. The average mutation rate begins to even out, and even increases slightly near the end of the simulation. The average learning rate remains close to zero.

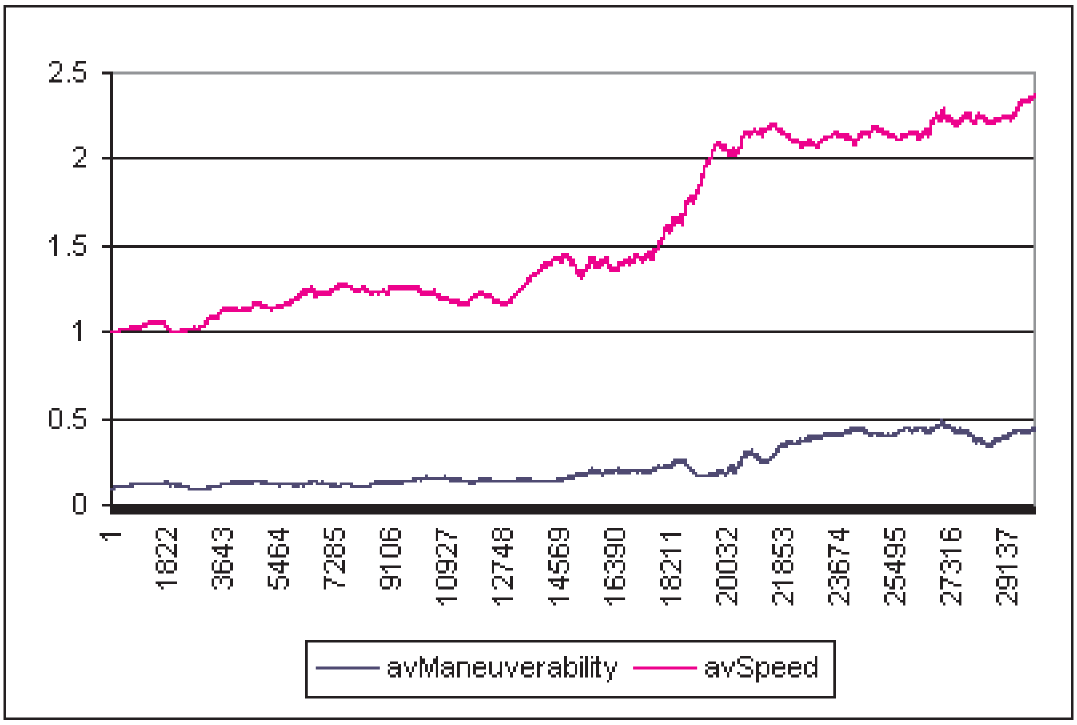


Chart of average maneuverability and speed, each in their respective units, for time steps 0 – 30000. Both of these values start increasing from the outset, though average speed experiences a sudden jump around time step 18211.

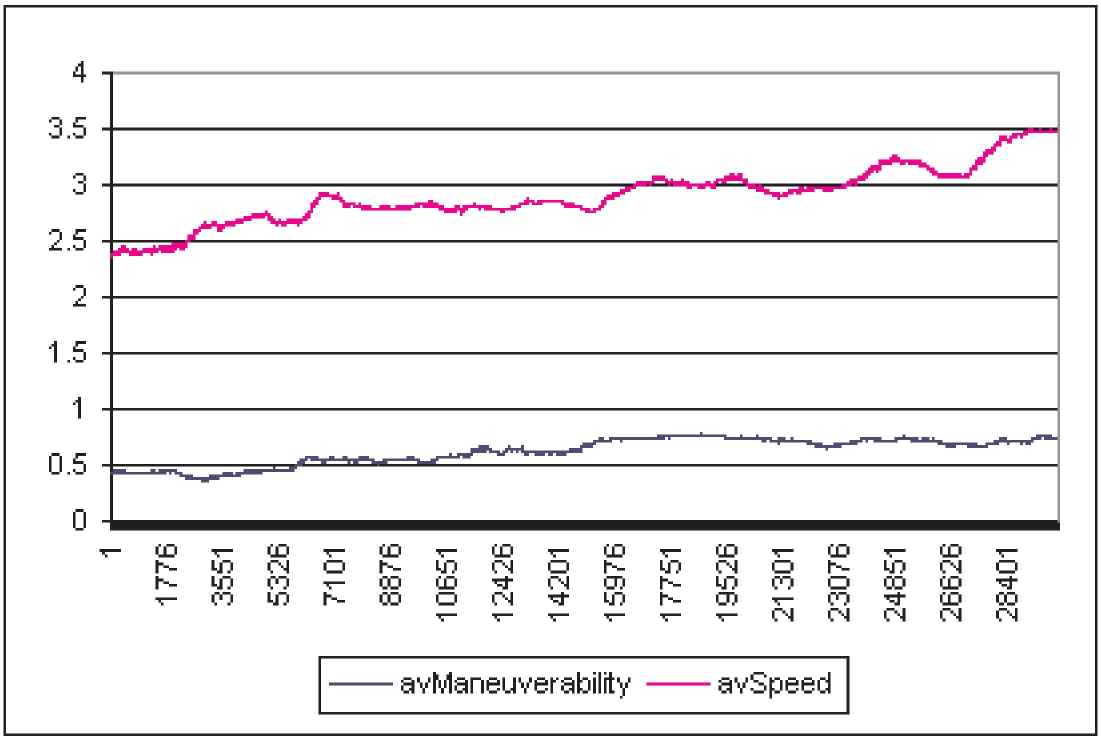


Chart of average maneuverability and speed, each in their respective units, for time steps 30001 – 60000. Both values continue to increase fairly gradually.

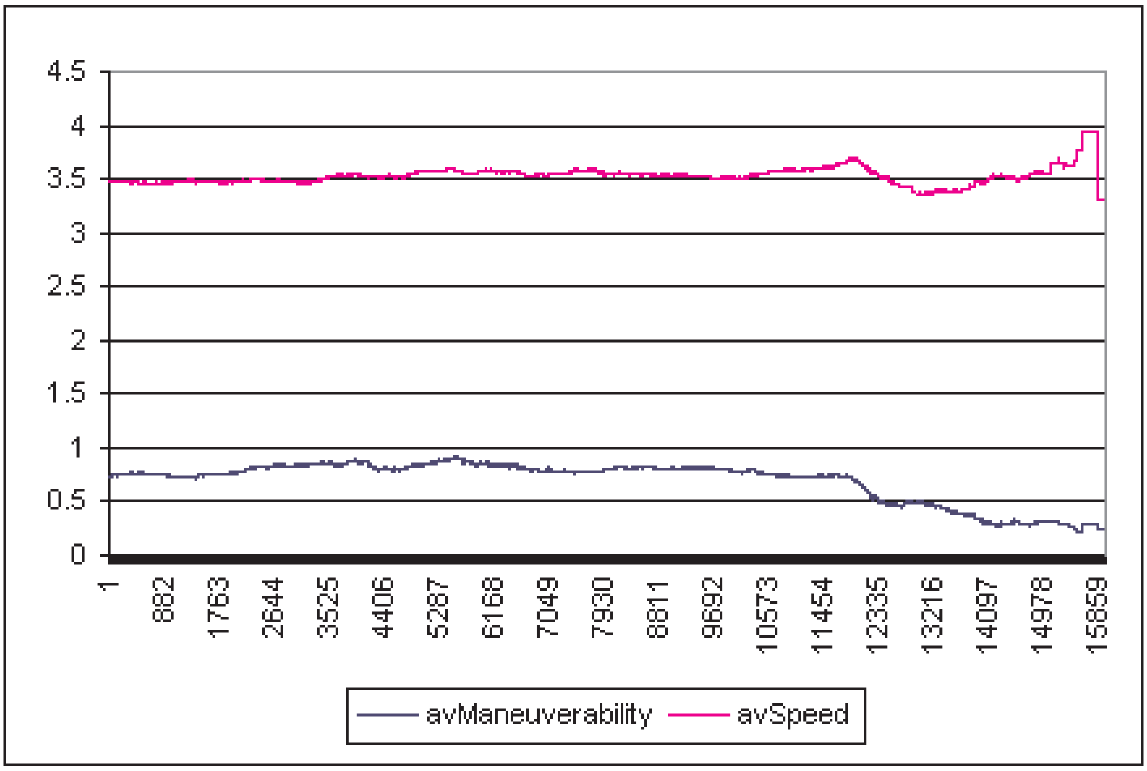


Chart of average maneuverability and speed, each in their respective units, for time steps 60001 – 75971. The average speed evens out at about 3.5. The average maneuverability evens out at just below 1, but begins to decrease again towards the end of the simulation.

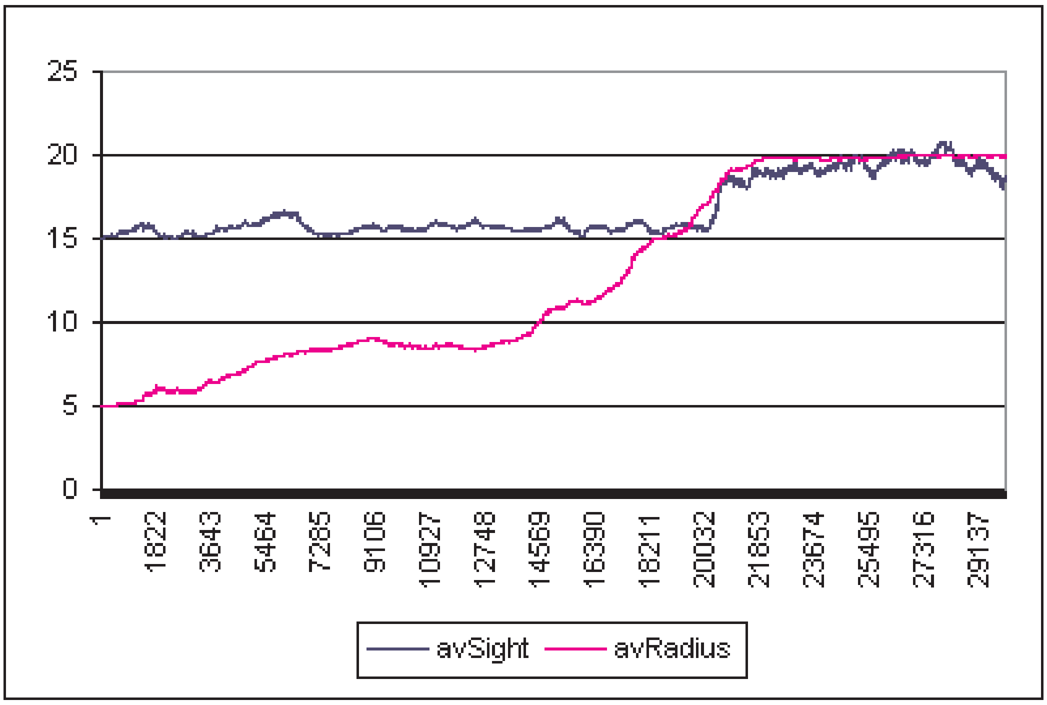


Chart of average sight range and radius, for time steps 0 – 30000. Average radius starts increasing from the beginning of the simulation, but the average sight range initially stays close to its starting value of 15. However, as the average radius continues to increase it soon reaches the maximum allowable radius of 20. As soon as the average radius surpasses the average sight range, the average sight range increases to 20 as well.

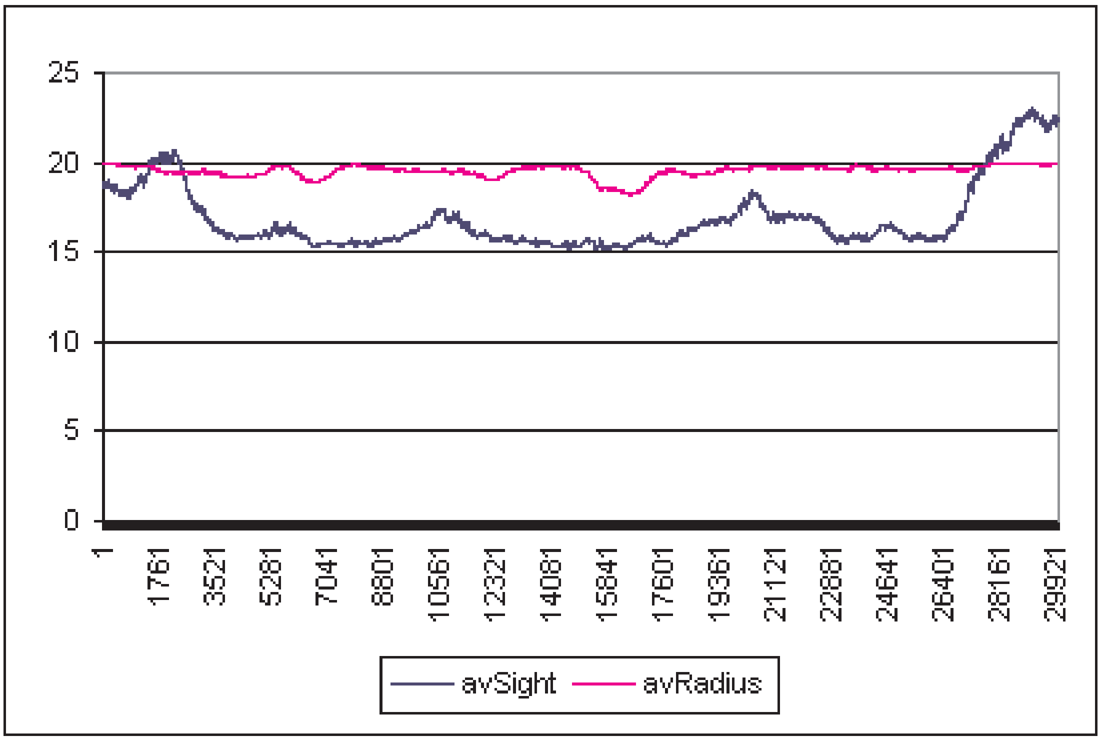


Chart of average sight range and radius, for time steps 30001 – 60000. The average radius stays close to 20. The average sight range drops back down near 15 and stays low for a while, but then picks up again to about 23.

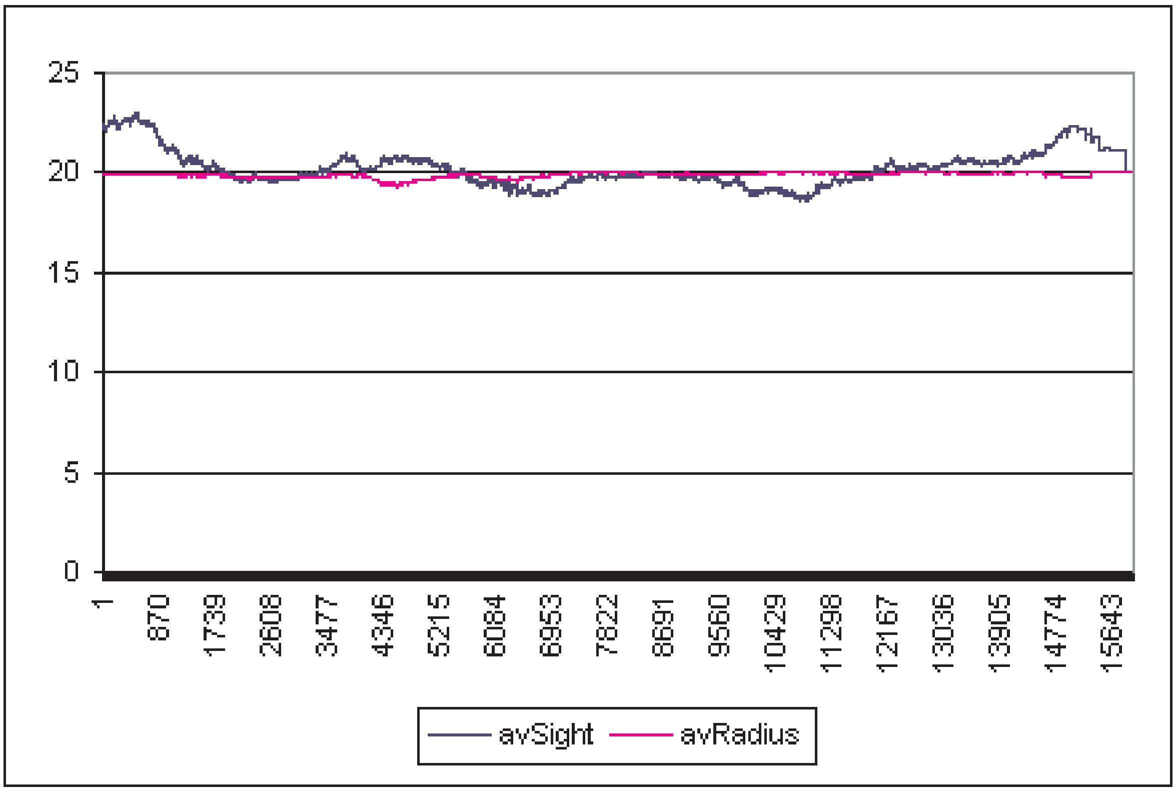


Chart of average sight range and radius, for time steps 60001 – 75971. The average radius continues to be about 20. The average sight range fluctuates a bit more, but stays fairly close to 20 as well.

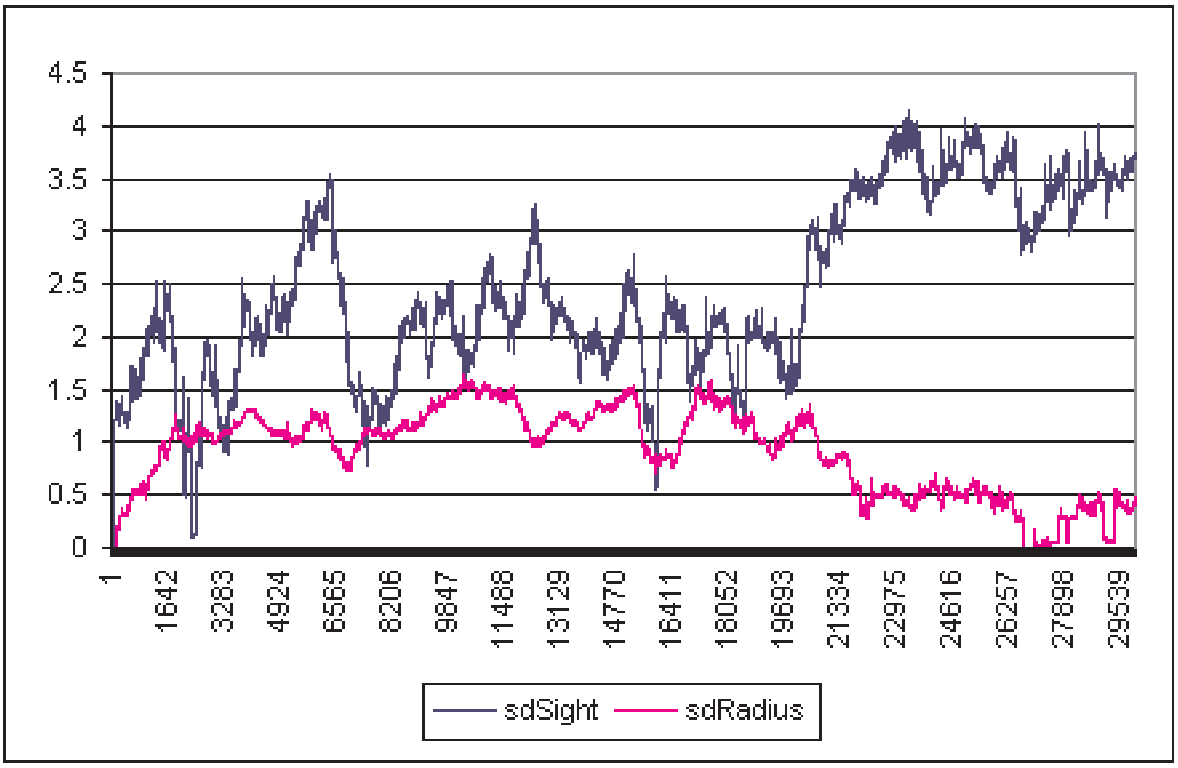


Chart of standard deviations of sight ranges and radii, for time steps 0 – 30000. Notice that the standard deviation between radii drops close to zero at about the time that the average radius is at its maximum value of 20. At the same time, the standard deviation between sight ranges increases greatly.

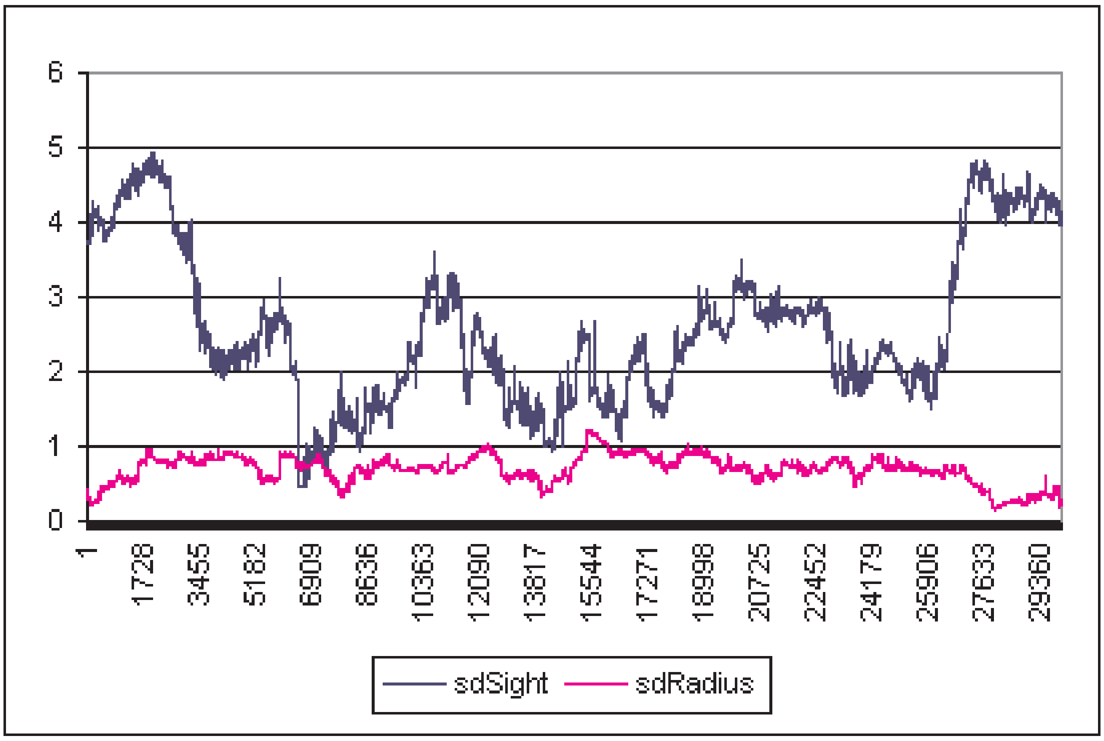


Chart of standard deviations of sight ranges and radii, for time steps 30001 – 60000. The standard deviation between radii stays low, implying that all animals have a radius close to 20. The standard deviation between sight ranges fluctuates wildly.

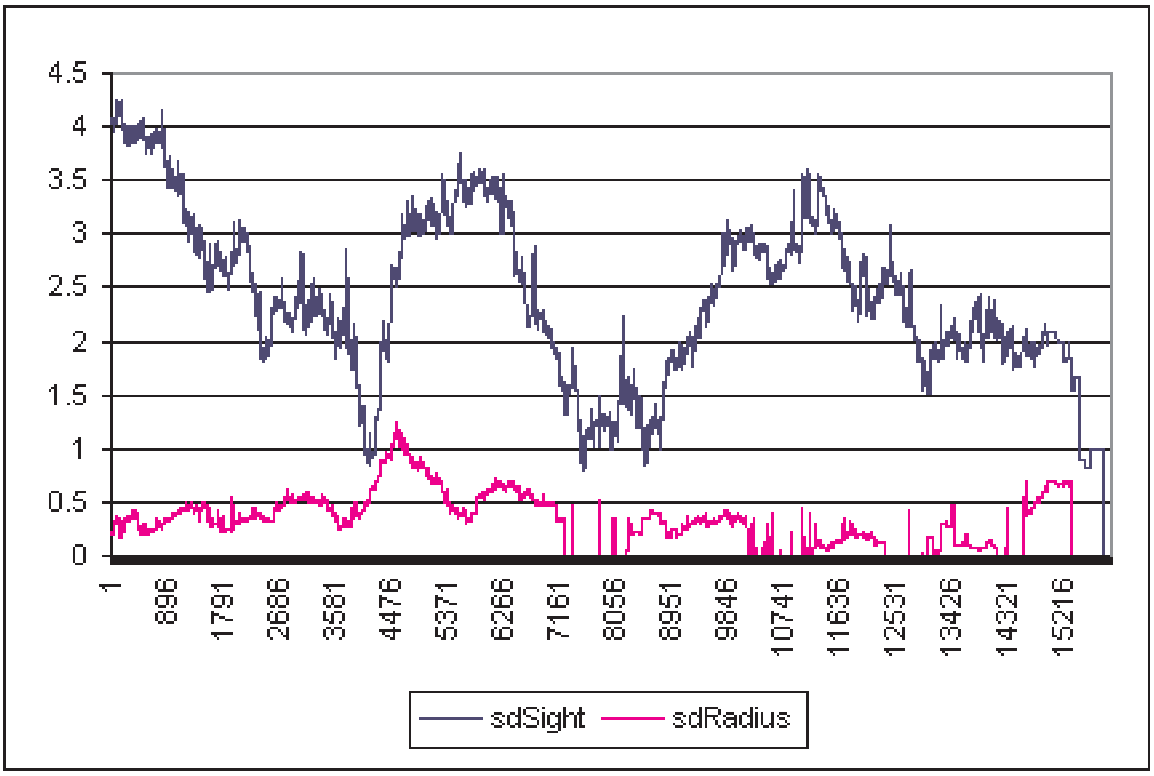


Chart of standard deviations of sight ranges and radii, for time steps 60001 – 75971. The standard deviation between radii continues to stay low. The standard deviation between sight ranges continues to fluctuate wildly.

3.1.2 Interpretation

As already mentioned, the data for this trial is reminiscent of plots for Volterra-Lotka predator-prey equations. The most general Volterra-Lotka system consists of two organisms: one predator and one prey. This system can be extended to three or more organisms, and the interactions can become more and more complex, but in the general form, as well as in certain cases of the specialized forms, the Volterra-Lotka equations exhibit behavior similar to that of our model, namely periodic fluctuations in several populations with the peaks of any predator species occurring slightly after peaks of the relative prey populations. We see this relationship between the entire population of animals (nearly all of which eat plants) and the plant population, and we also see it between the herbivore population and the herbivore-carnivore population. The purpose of the Volterra-Lotka equations is to model real world predator-prey interactions, and there are instances of predator-prey systems in the real world that have been shown to roughly correspond with this model, the most famous of which is likely the lynx-snowshoe hare model.

The popularity of plant eating over all other eating types can likely be traced primarily to the following two features of the simulation's design: the action order precedence, the regular growth and therefore availability of plants. Given the possibility, an animal is programmed to always choose a plant over any other food source. This was done because it was easier than coding for the possibility of an animal choosing between multiple food sources either randomly, based on anticipated energy gain, through yet another neural network, or through some other means. Of course, eating plants is also popular because it is the only food source that is guaranteed to replenish itself. Carnivores, cannibals and carrion eaters are all at risk of depleting their food source to the point that it never regenerates. Indeed, this appears to be exactly what happened in this trial. The herbivore-carnivores hunted the regular herbivores to extinction, and then died out themselves, in spite of their own ability to eat plants. According to the data, the average energy of all organisms in the last few time steps of this trial equaled the average maximum energy. This seems to imply that several old predators close to death finished off the presumably younger generation of herbivores and then died with full energy as they continued to eat plants.

It is of course unfortunate that the population died out. The emergence of both a predator population and a prey population demonstrates the simulation's ability to support the emergence of diversity, and the Volterra-Lotka behavior demonstrates the simulation's ability to produce complex behavior that is similar to real world models, although it was not explicitly coded into the simulation.

Other interesting occurrences include the quick emergence and extinction of carrion eaters. The initial success of the carrion eaters, and the quickly following inability of the world to sustain them, can likely be explained by another quirk of the simulation. The initial population of animals all have the same initial energy. This energy amount in turn determines each animal's maximum energy and death point. In this case the initial energy was 200 for each animal, which is much more than what each child born from this generation started with. The high initial energy drove the maximum energy of these animals up, and the initial lack of predators and abundance of plants enabled most of these animals to easily survive until reaching the end of its lifespan, which was also the same for all of these animals. Therefore nearly all of the initial population passes away on the same time step. Furthermore, their high initial energy levels drove their death points up, and when an animal dies of old age, its energy is set to its death point. Carrion eaters are initially successful in the world because there is a point near the start when all of the initial population dies, thus filling the world with many energy rich corpses to eat. Of course, this is a one time occurrence, so the number of carrion eaters sustainable by the world drops after the corpses of the initial generation are gone.

Also seen in the data are the changes in the mutation and learning rates over time. The mutation rate starts at 1 to encourage new possibilities. There is no where to go from 1 except down, so it is only natural that the average mutation rate decreases over time. The change in the average learning rate was somewhat surprising however. Low learning rates seem to be strongly selected for. Is learning bad? Or is learning so sensitive that it is only an advantage when it is gradual, and thus the learning rate is small. Given that the learning rate is used to modify synaptic weights on every time step, it is reasonable to assume that a large learning rate would have an animal completely changing its behavior on every time step, and thus being completely unable to act effectively, but results from the next trial contradict this.

The last bit of data that is important in interpreting this trial is the change in the average radius over time. Comparing the plots of the average radius and the herbivore population over time, we see that there is a drop in the herbivore population around where the average radius increases to its maximum value of 20. The standard deviation between animal radii also drops close to 0 at this point, meaning that only large animals remain. This is connected to the way that predation is modeled. An animal's ability to eat another depends on relative size (radius), such that a predator can only be successful if it is bigger than its prey. Admittedly, this is a very synthetic feature, and there are many real world examples which contradict it. The consequence of it within the simulation is that animals with large radii are strongly selected for. Of course, the increase in predator size leads to an increase in prey size, because only larger prey organisms can avoid being eaten by the predators. The result is that the whole population rapidly approaches the artificially imposed maximum value for animal radii. The herbivore population never fully recovers from the size based pruning that the large herbivore-carnivores induce, and the eventual downfall of the population as a whole could perhaps be traced back to this moment. Being large does of course have disadvantages, such as increased movement costs and decreased maneuverability. These features were meant to give some balance to the radius trait, but animals tend towards being large in spite of them.

3.2 Trial 2: Popularity of Carrion Eating

This trial was allowed to run for 70179 time steps before being manually terminated. The population showed little sign of change or fluctuation. It cannot be proven that this population would have continued to live (anymore than one can prove the continued existence of humanity), but the population seems to have been stable. This stability was likely due to the lack of predators in the world. What makes this trial interesting is the large number of herbivore-carrion eaters that evolved in the population. The resulting dynamics are very different from those of trial 1. There is a distinct lack of periodically fluctuating behavior. Besides the herbivore-carrion eaters there are also many pure herbivores. Other eating behaviors briefly appear in environment, but seem to have no effect on the long term behavior of the trial. Also interesting are the learning rate, mutation rate and radius traits, which in this trial behave very differently in comparison with their behavior in trial 1. The mating cost, starting energy and death point are also of interest in this trial, because they are closely linked with when an animal dies, which is important to carrion eaters. Mating cost and starting energy together determine the death point of each animal. Their importance is discussed in the interpretation of this data.

3.2.1 Data

interactionDistance = 10	minMatingTime = 0	growthTimeStep = 5
speciesEqualityConstant = 0.15	maxMatingTime = 7	plantsPerGrowth = 100
minStartEn = 0.1	sStartEnergy = 0.5	plantNutrition = 65
maxStartEn = 0.9	sSight = 0.0	binGridSizeX = 3
minSight = 15	sHerbivore = 0.65	binGridSizeY = 3
maxSight = 70	sCarnivore = 0.35	binWidth = 100
minManeuverability = 0.1	sCannibal = 0.35	binHeight = 100
maxManeuverability = 3.0	sCarrion = 0.35	initialPopulation = 150
maneuverabilityRestriction = 0.1	sManeuverability = 0.0	initialPlants = 200
minMoveSpeed = 1.0	sSpeed = 0.0	netArchitecture = [26,26,26,2]
maxMoveSpeed = 5.0	sLifespan = 0.3	maxFloatMutation = 0.3
minMove = 0.2	sMutation = 1.0	maxIntMutation = 3
moveCostMultiple = 0.05	sRadius = 0.0	initialEnergy = 200
minMaturity = 30	sMateCost = 0.2	metabolicRate = 0.05
maxMaturity = 70	sLearning = 0.1	decay = 0.3
minLifespan = 300	sMaturity = 0.5	seed1 = 17235
maxLifespan = 900	sMateTime = 0.5	seed2 = 27373
minMutationRate = 0.1	sDummy = 0.5	seed3 = 34189
maxMutationRate = 1.0	sMateRecovery = 0.5	seed4 = 57123
minRadius = 5	deathEnergy = 0.75	seed5 = 16529
maxRadius = 20	energyRestriction = 25	seed6 = 24917
minMatingCost = 50	predationMultiplier = 18	seed7 = 27712
maxMatingCost = 300	predationConstant = 150	seed8 = 82144
minMatingRecovery = 40	carrionMultiplier = 18	seed9 = 56728
maxMatingRecovery = 90	cannibalGracePeriod = 40	seed10 = 33221

Constants for trial 2

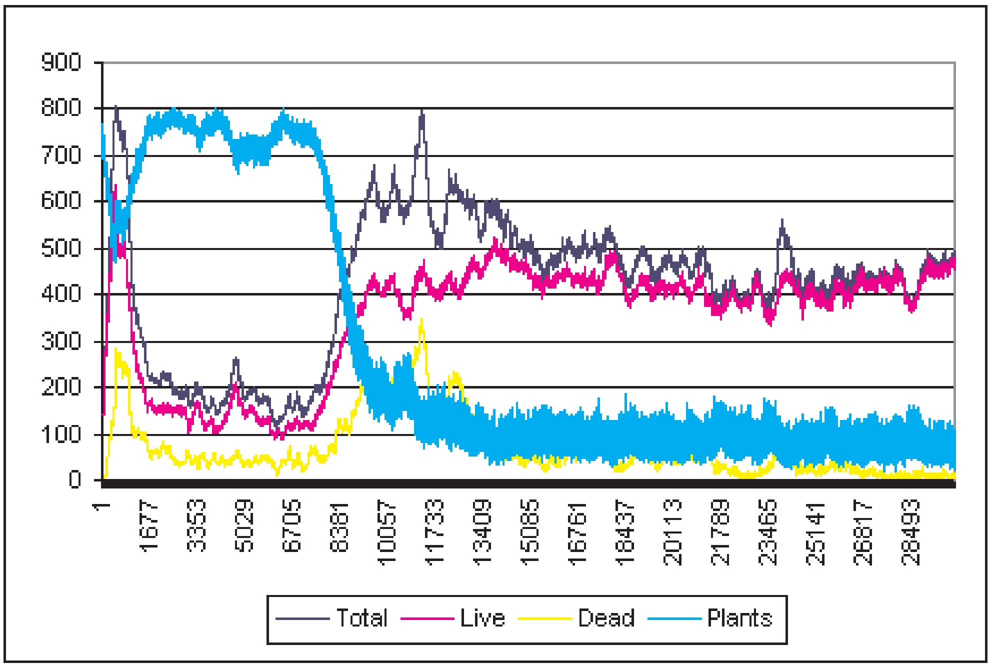


Chart of the total animals, live animals and dead animals, along with the number of plants, for time steps 0 – 30000. After suffering a sharp decline early in the simulation, the animal population quickly recovers and then remains almost constant for the rest of this time period. At the same time the plant population plummets and remains very low. It is also interesting to note that the number of corpses in the world is very small after the point of the animal population's increase. Nearly all the animals in the world are living.

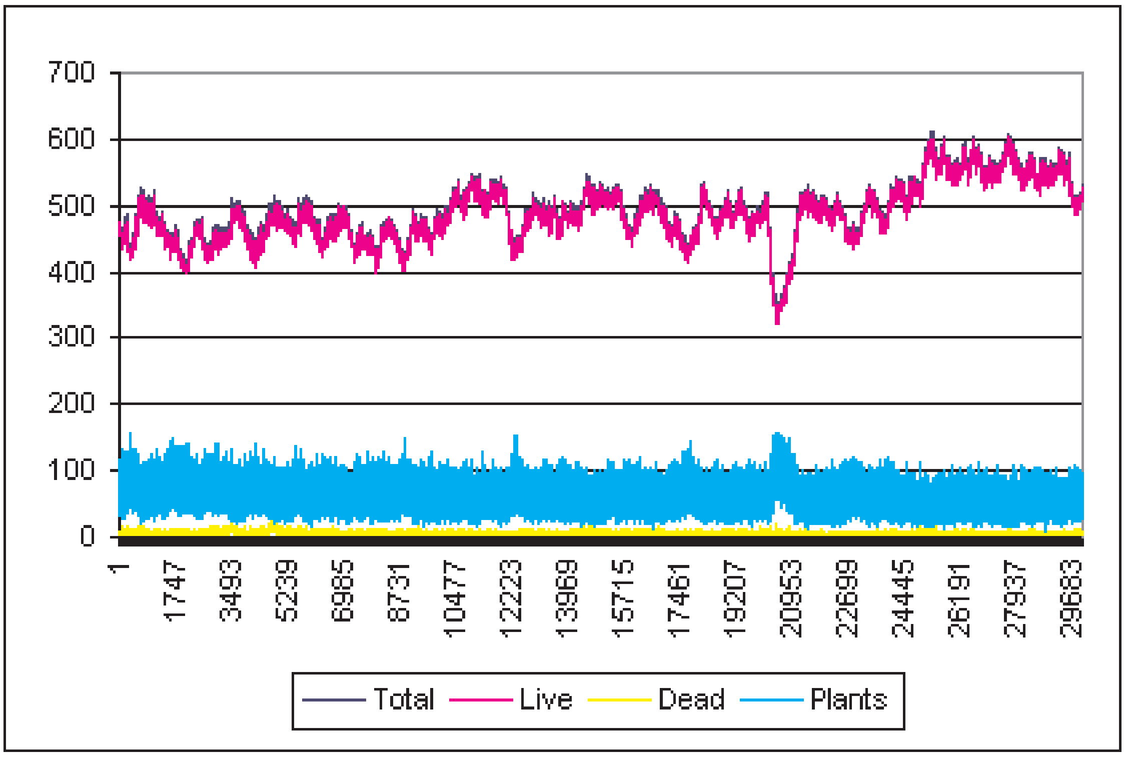


Chart of the total animals, live animals and dead animals, along with the number of plants, for time steps 30001 – 60000. The plot for “Live” and “Total” animals is nearly identical because the number of “Dead” animals is so small. The plant population is also extremely small. All of these values remain fairly stable. The animal population grows a very small amount near the end of this timeframe.

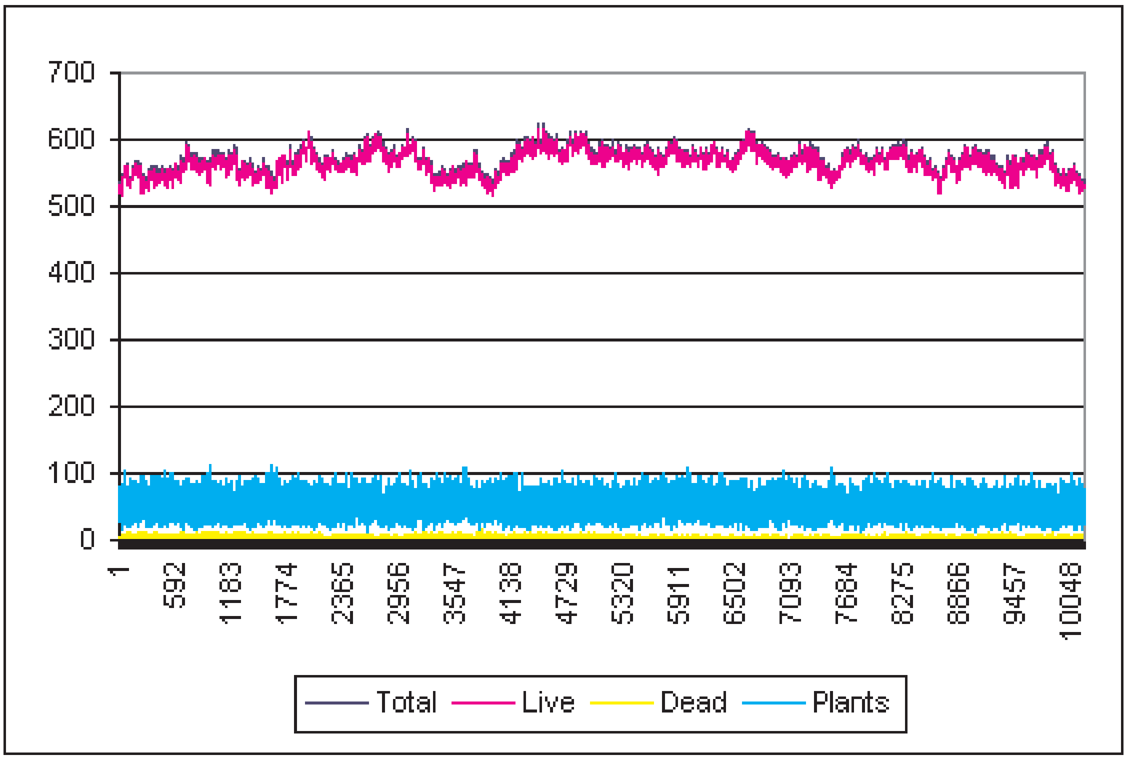


Chart of the total animals, live animals and dead animals, along with the number of plants, for time steps 60001 – 70179. All plots remain stable. This is unusual, and its implications are interesting. However, it is not interesting to observe, therefore the trial is terminated at time step 70179.

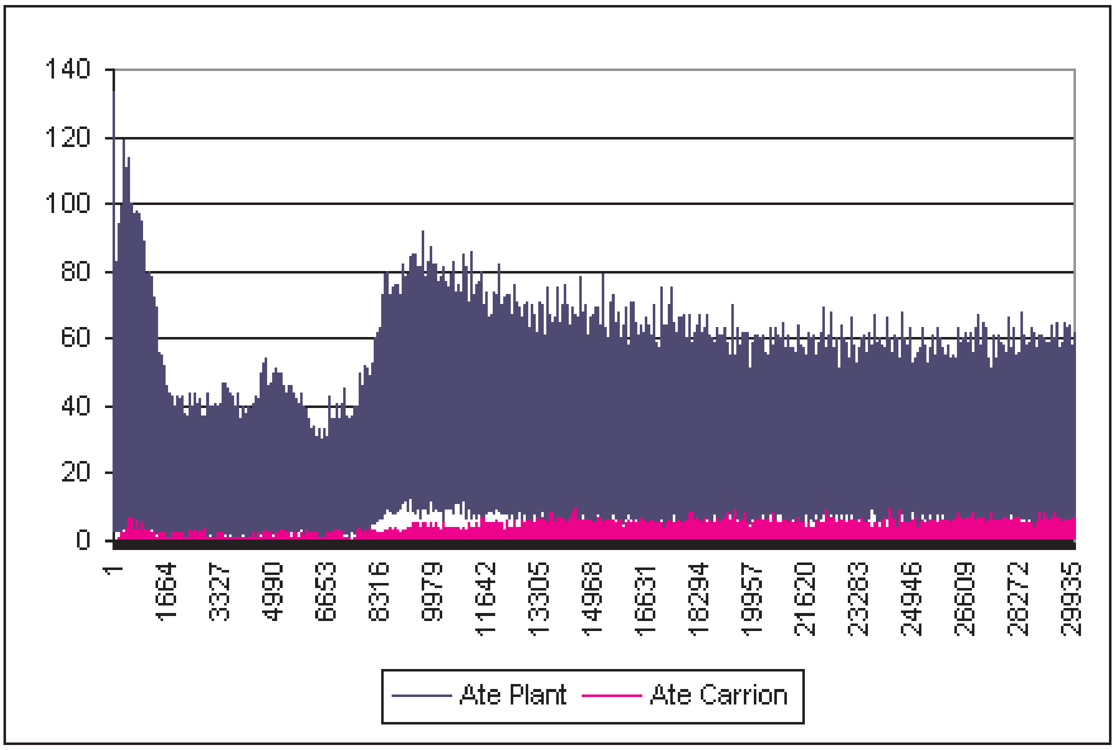


Chart of the number of animals that ate plants and those that ate carrion, for time steps 0 – 30000. The number of animals that eat plants remains both large and consistent after an early drop which corresponds to the time when the animal population itself was particularly low. Likewise the number of animals that eat carrion is fairly consistent after the early population drop.

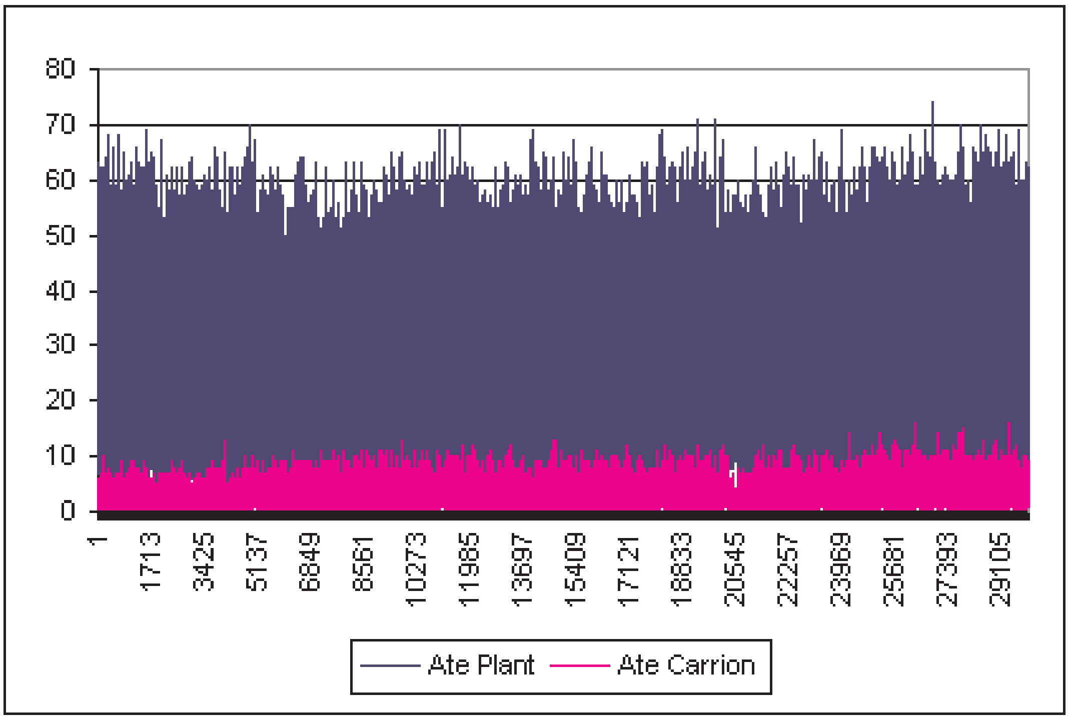


Chart of the number of animals that ate plants and those that ate carrion, for time steps 30001 – 60000. The number of animals eating plants remains large and stable. The number of animals eating carrion increases very slowly over time.

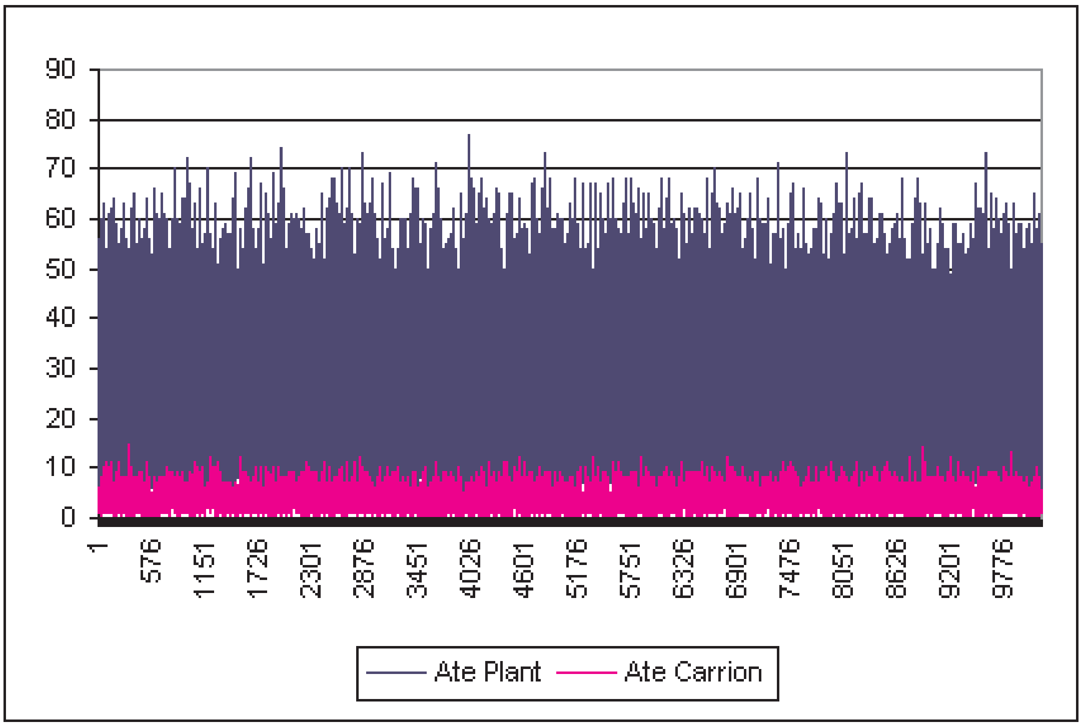


Chart of the number of animals that ate plants and those that ate carrion, for time steps 60001 – 70179. The number of animals eating plants and carrion continues to stay the same.

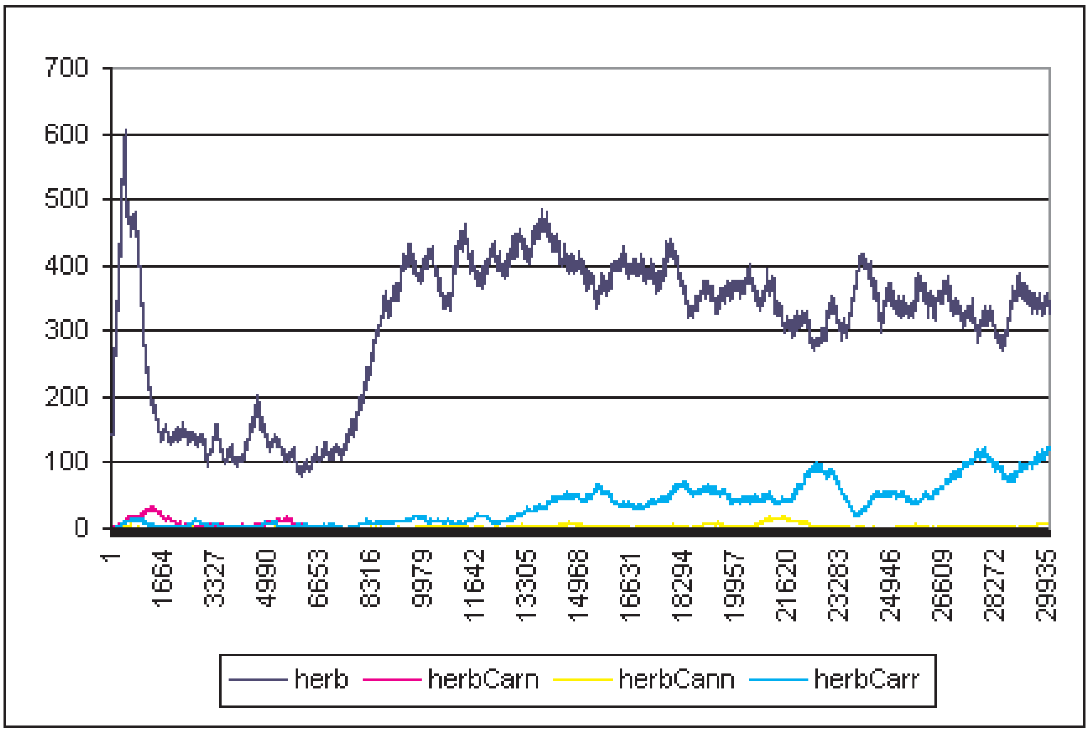


Chart of the number of herbivores, herbivore-carnivores, herbivore-cannibals and herbivore-carrion eaters, for time steps 0 – 30000. At first the number of herbivores in the population is nearly identical to the number of living animals in the population, though herbivore-carnivores, herbivore-cannibals and herbivore-carrion eaters are all present in small numbers early on. The herbivore-carnivores die out in the population drop near the start of the trial. After the herbivore population starts to recover, the herbivore-carrion eater population also starts to rise. Very small numbers of herbivore-cannibals continue to appear.

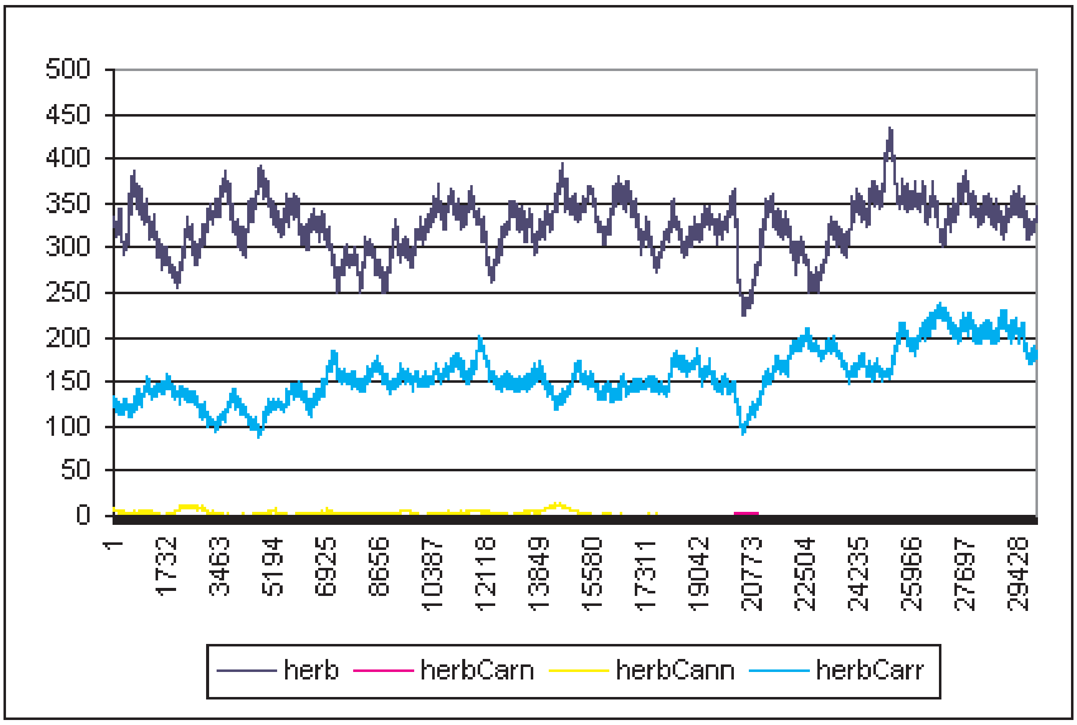


Chart of the number of herbivores, herbivore-carnivores, herbivore-cannibals and herbivore-carrion eaters, for time steps 30001 – 60000. Herbivore-carnivore and herbivore-cannibal numbers remain insignificant. The herbivore population remains large, though fluctuates quite a bit. Meanwhile the herbivore-carrion eater population grows to more than half the herbivore population.

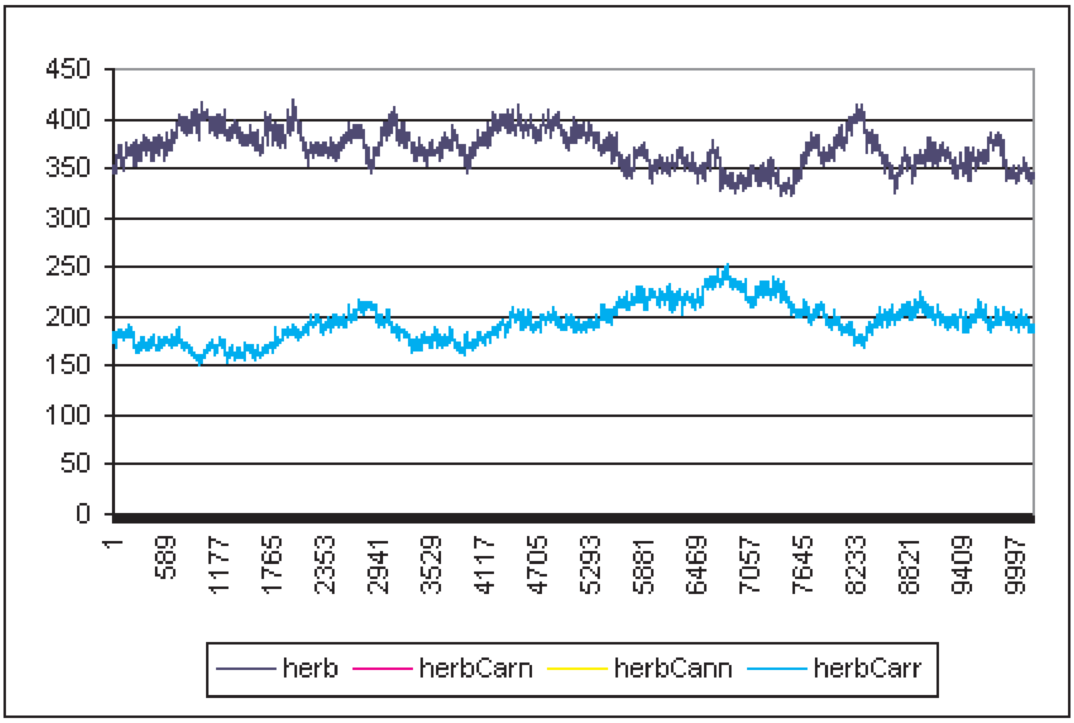


Chart of the number of herbivores, herbivore-carnivores, herbivore-cannibals and herbivore-carrion eaters, for time steps 60001 – 70179. Herbivore-carnivores and herbivore-cannibals have died out completely. The herbivore population remains large (around 400) and the herbivore-carrion eater population continues to be slightly more than half the pure herbivore population (slightly over 200).

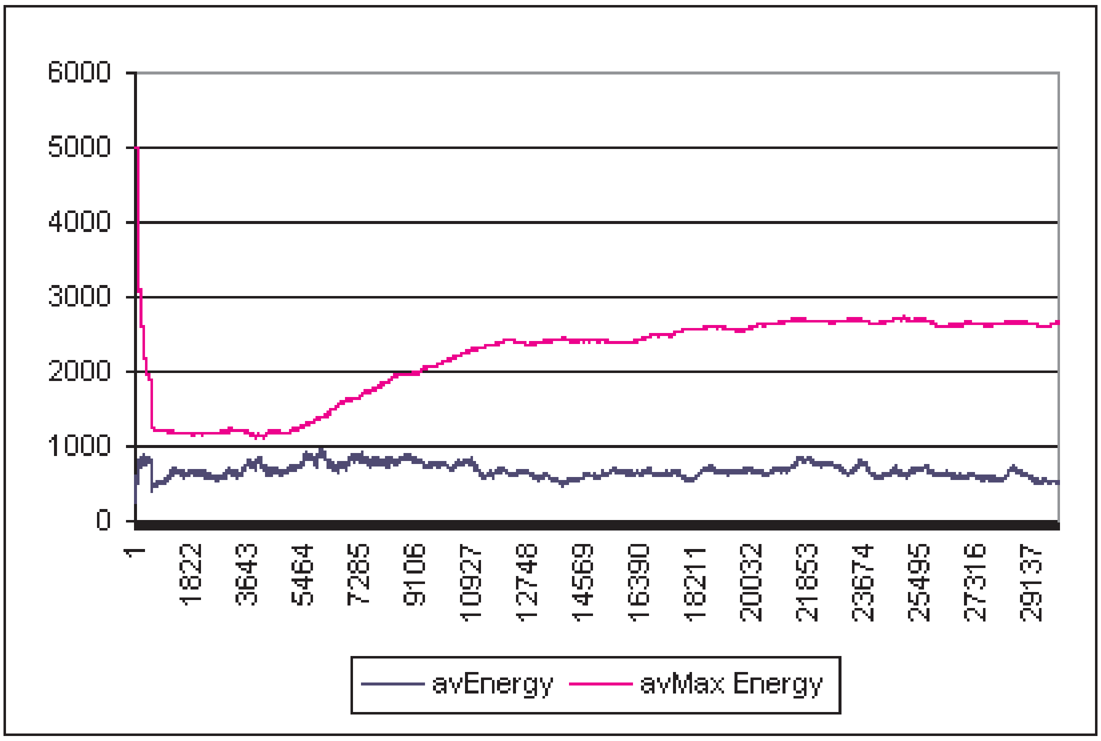


Chart of the average energy level and average maximum energy level, for time steps 0 – 30000. The average maximum energy level picks up at the point where the animal population recovers from its decline. It is however interesting to note that the average energy level remains about the same when the average maximum energy level increases.

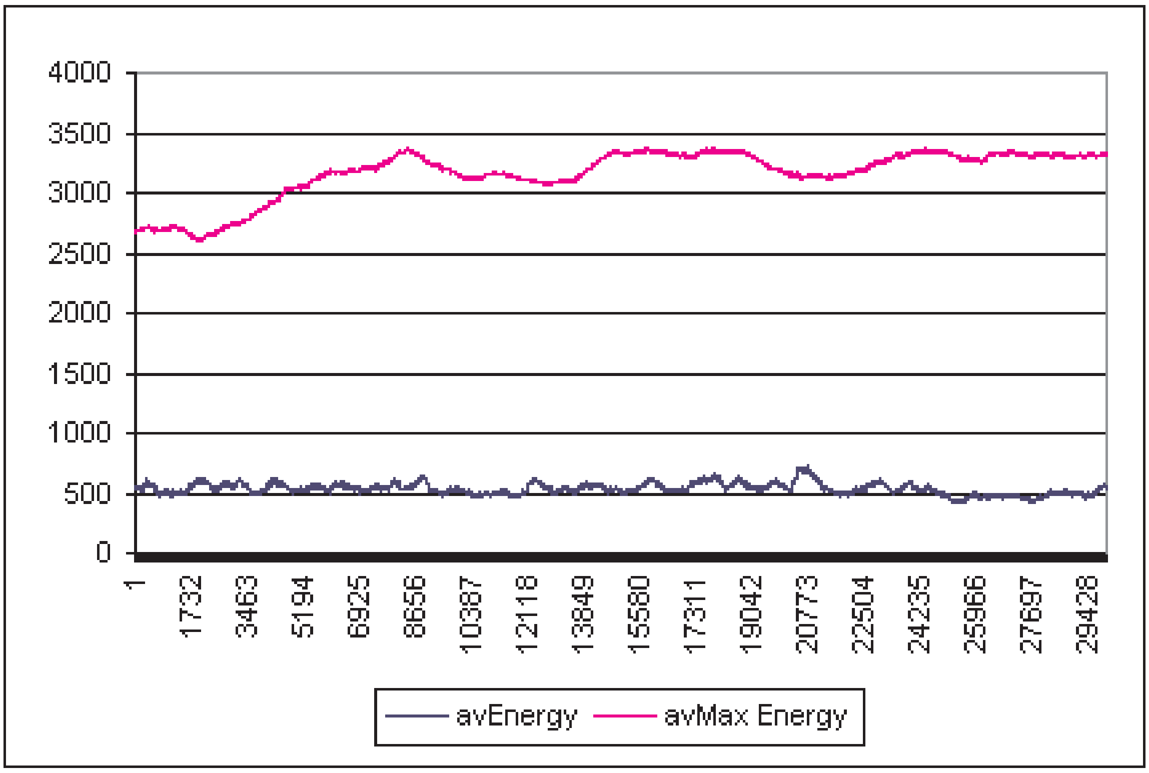


Chart of the average energy level and average maximum energy level, for time steps 30001 – 60000. The average energy level stays close to 500 even as the average maximum energy level continues to grow.

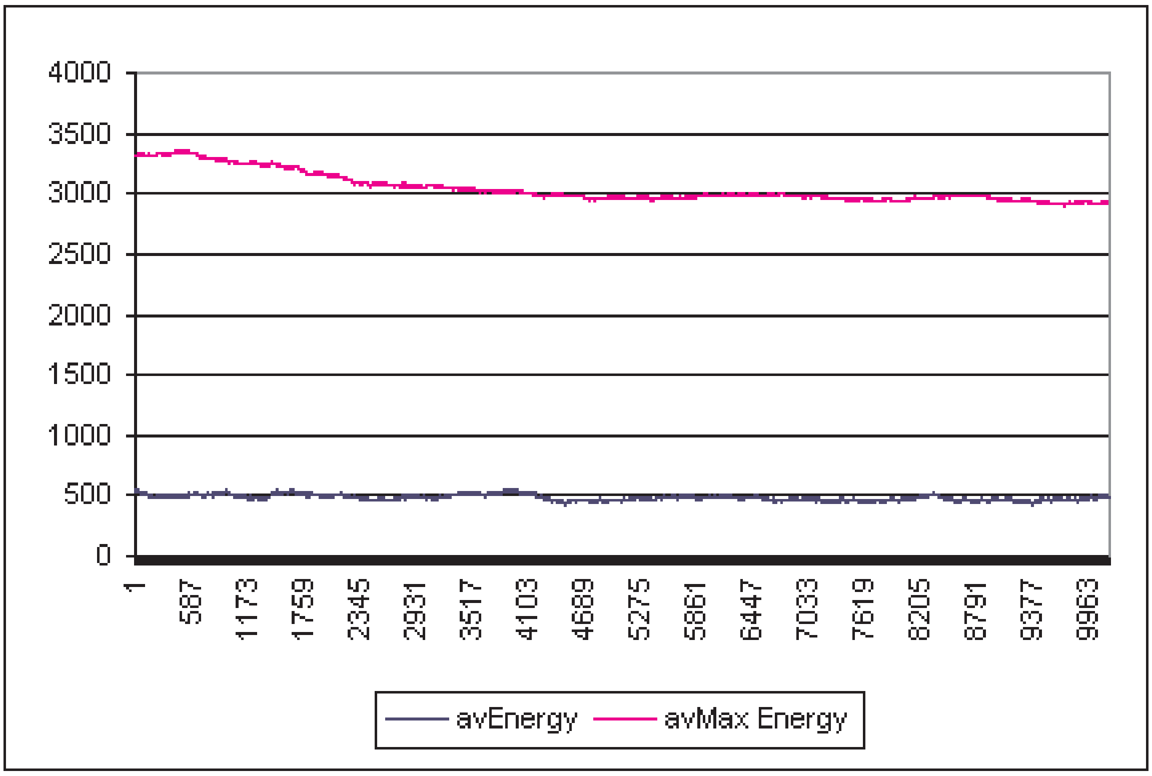


Chart of the average energy level and average maximum energy level, for time steps 60001 – 70179. The average energy level fluctuates even less, staying very close to 500. The maximum energy decreases very slightly, but remains large.

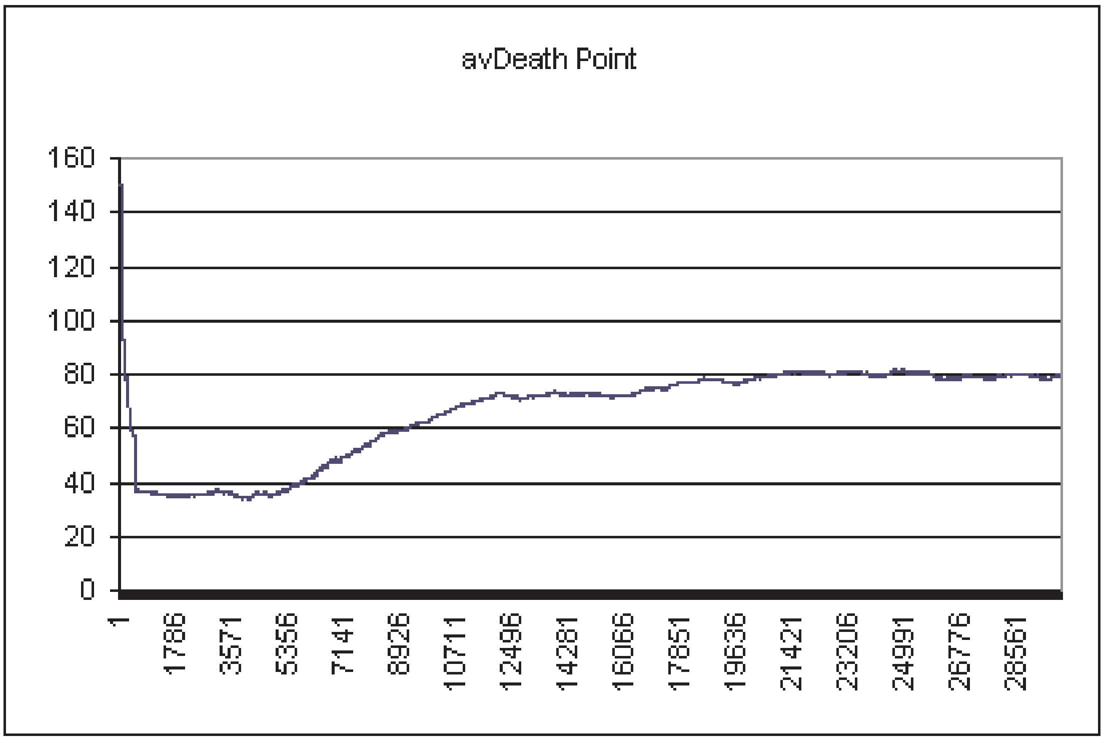


Chart of the average death point, for time steps 0 – 30000. The death point and maximum energy level are both derived from the mating cost and start energy traits, so all fluctuations of the average death point correspond to qualitatively identical fluctuations of the average maximum energy level. The only difference is scale. Therefore the average death point increases as the population comes out of its initial decline.

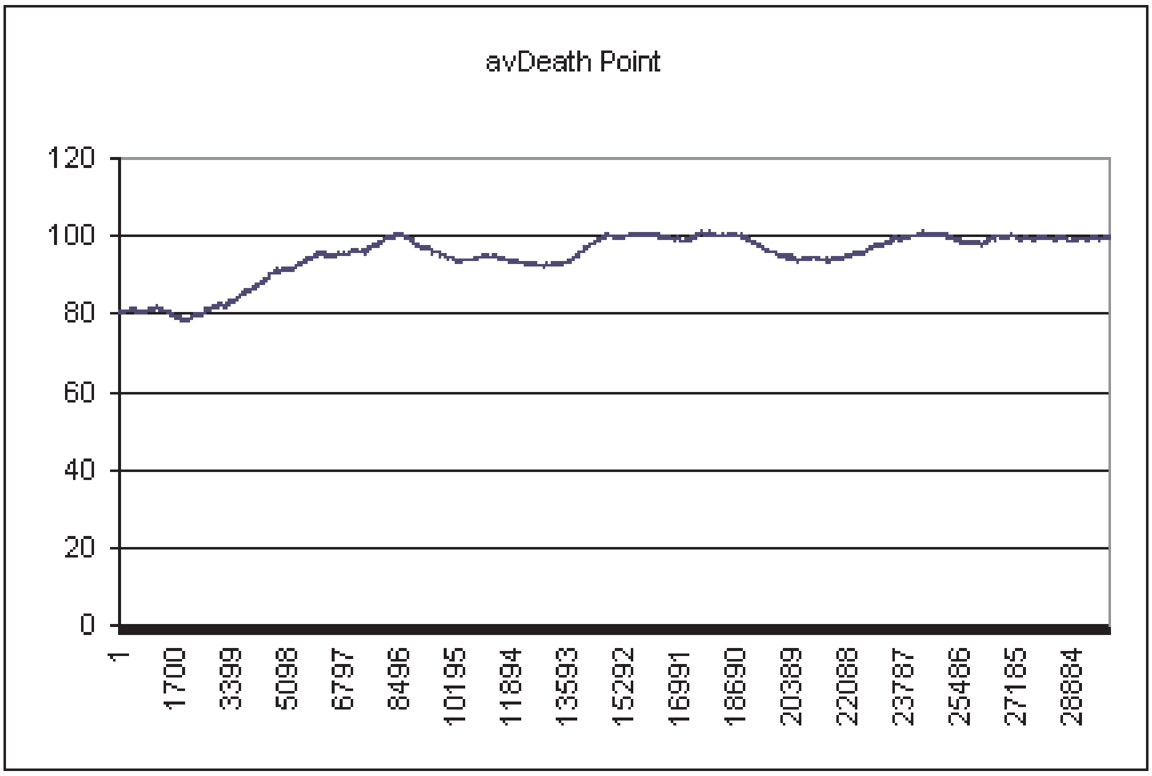


Chart of the average death point, for time steps 30001 – 60000. As with the average maximum energy level, the average death point increases only slightly in this time range.

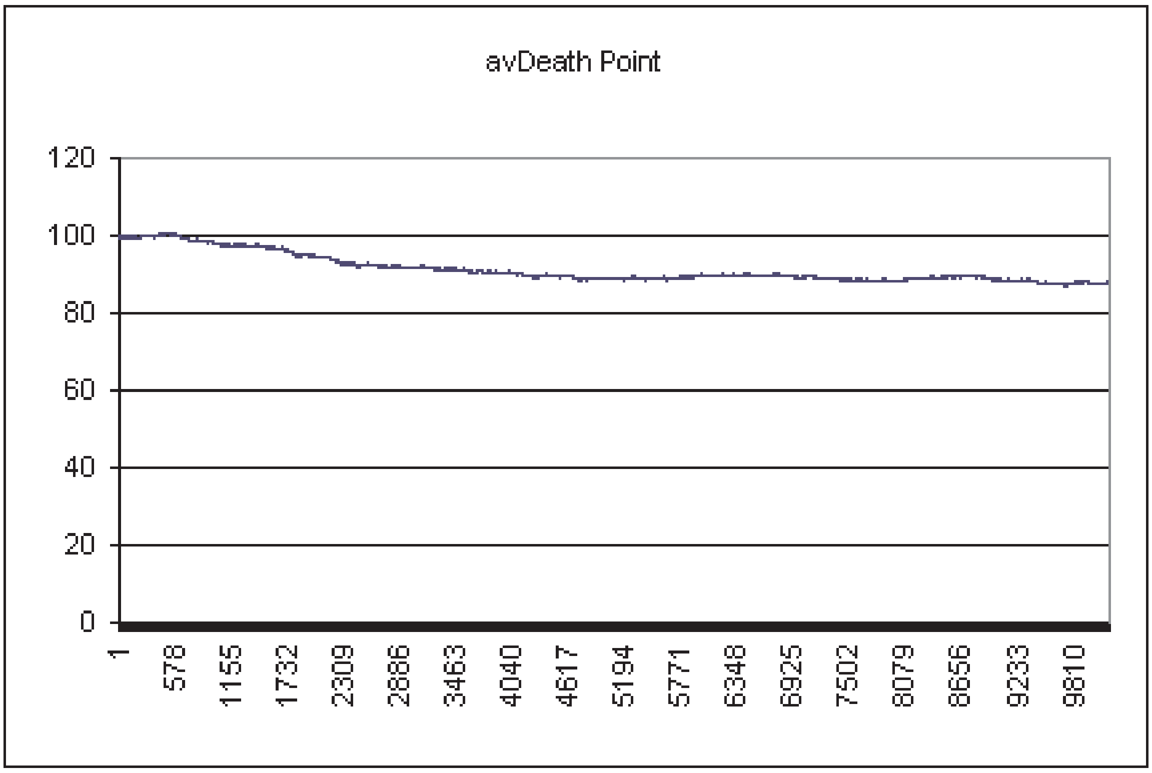


Chart of the average death point, for time steps 60001 – 70179. Here the average death point decreases slightly, but does not reach any lower than it was at time step 30001. It remains stable.

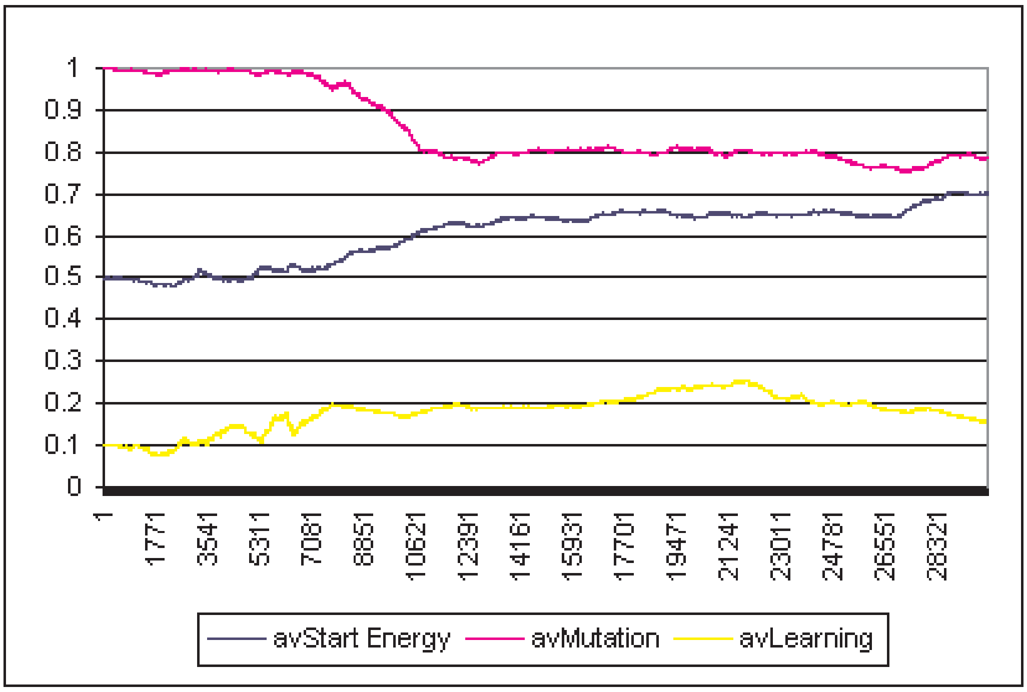


Chart of the average starting energy, average mutation rate and average learning rate, for time steps 0 – 30000. The average starting energy steadily increases throughout this time interval, which explains the increase in both the average maximum energy level and average death point. The average mutation rate starts steady at 1, but then drops quickly to 0.8 around the time that the animals population recovers from its decline. From this point on the average mutation rate remains stable. The average learning rate starts increasing early and settles at about 0.2.

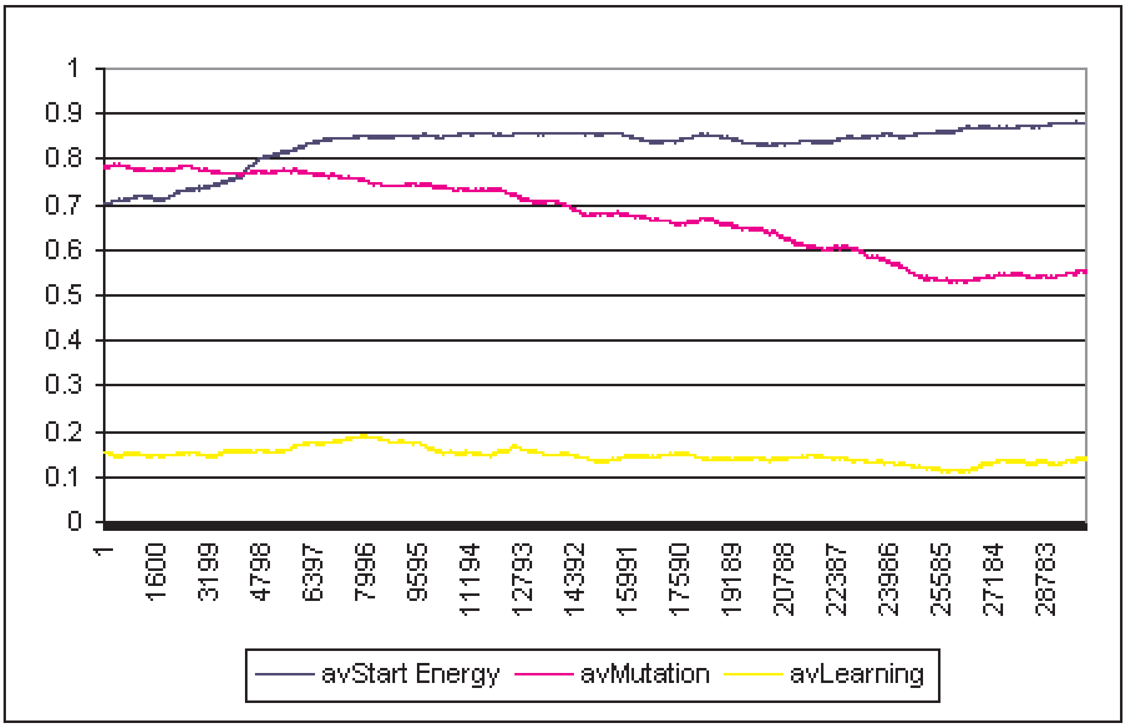


Chart of the average starting energy, average mutation rate and average learning rate, for time steps 30001 – 60000. The average learning rate hovers slightly below 0.2. The average starting energy approaches its maximum value of 0.9 and stays close to it. The average mutation rate drops even more, and seems to even out slightly at about 0.55.

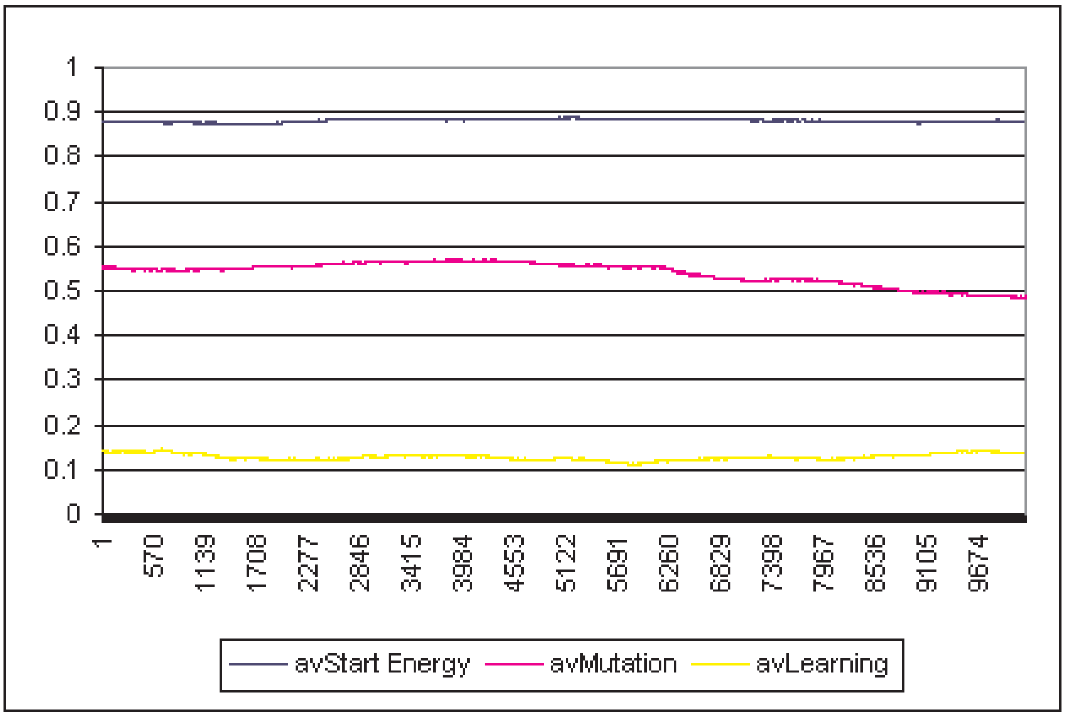


Chart of the average starting energy, average mutation rate and average learning rate, for time steps 60001 – 70179. The average starting energy remains close to its maximum value of 0.9. The average learning rate also remains stable between 0.1 and 0.2. The average mutation rate stays at about 0.55 for a while before decreasing even more to below 0.5. This decrease in mutation rate was another reason for terminating the trial. With a decreasing mutation rate the simulation was exceedingly less likely to produce any new or interesting behavior.

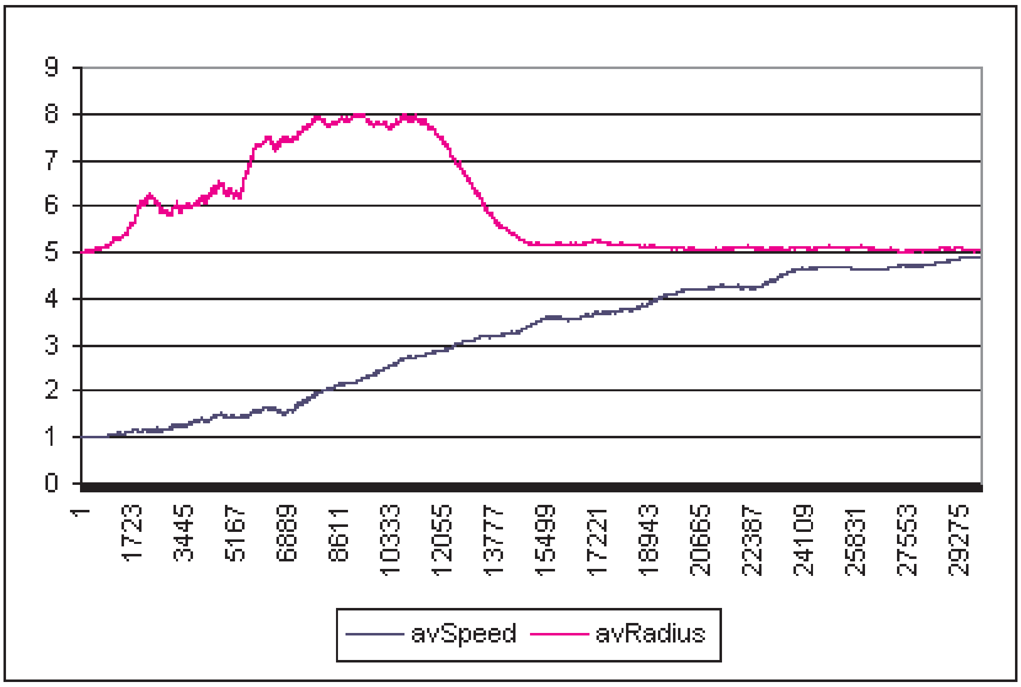


Chart of the average speed and average radius, for time steps 0 – 30000. The average radius starts at the minimum value of 5. Mutations cause this average value to increase, since it is only possible for the radius to increase from its minimum value. However, the simulation seems to select strongly against larger radii, so the average value returns to the minimum of 5. Curiously, the average speed increases steadily throughout this time interval.

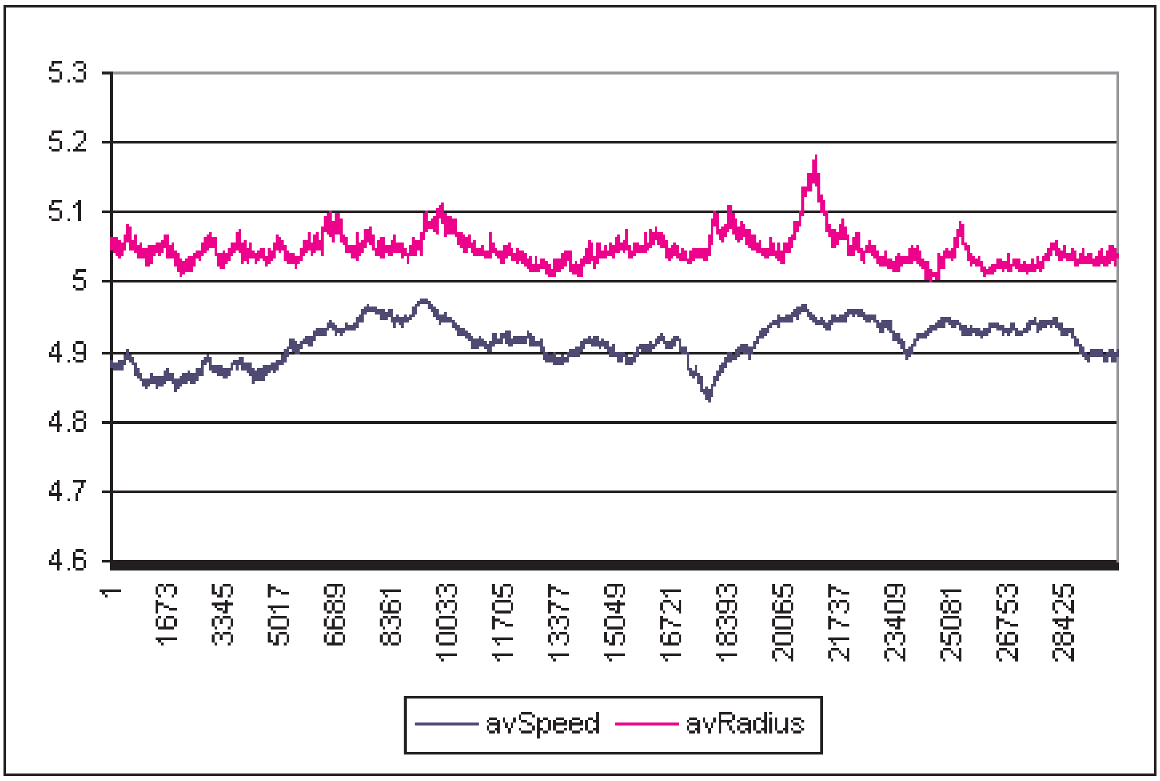


Chart of the average speed and average radius, for time steps 30001 – 60000. The average radius remains comfortably stable slightly above 5, and the average speed remains comfortably stable around 4.9.

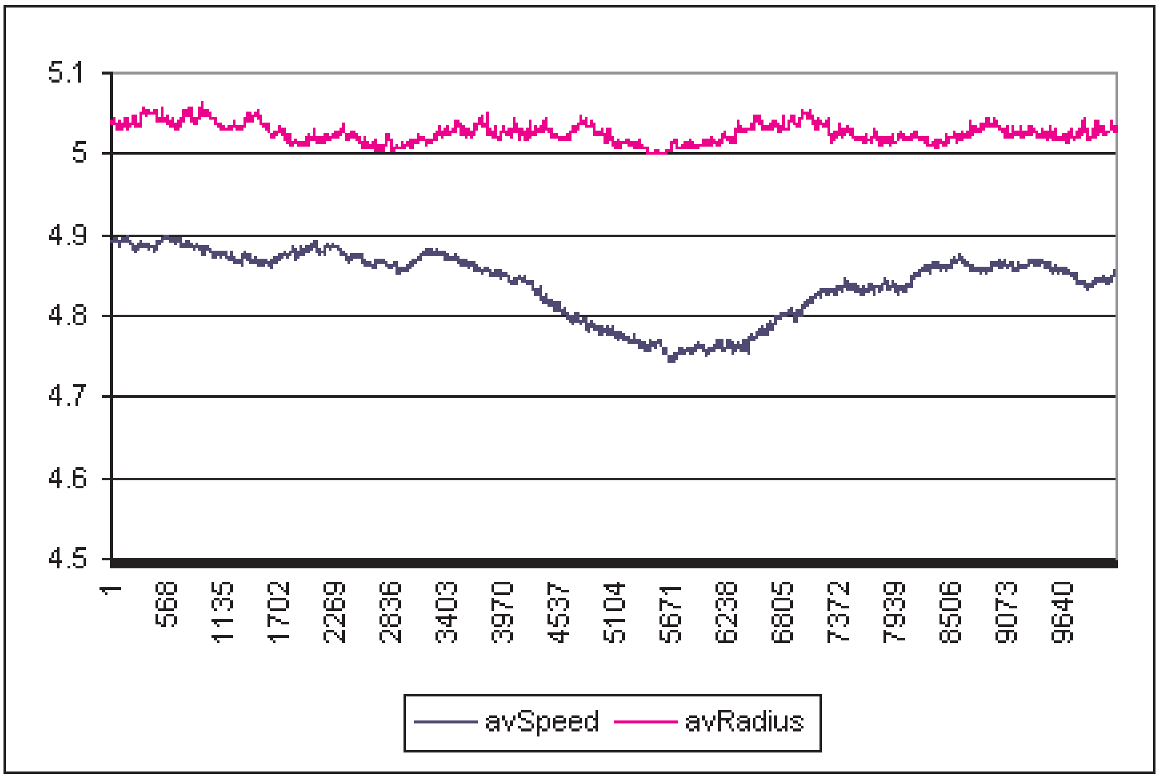


Chart of the average speed and average radius, for time steps 60001 – 70179. The average radius remains close to 5. The average speed decreases slightly to about 4.75, but then increases again to about 4.85.

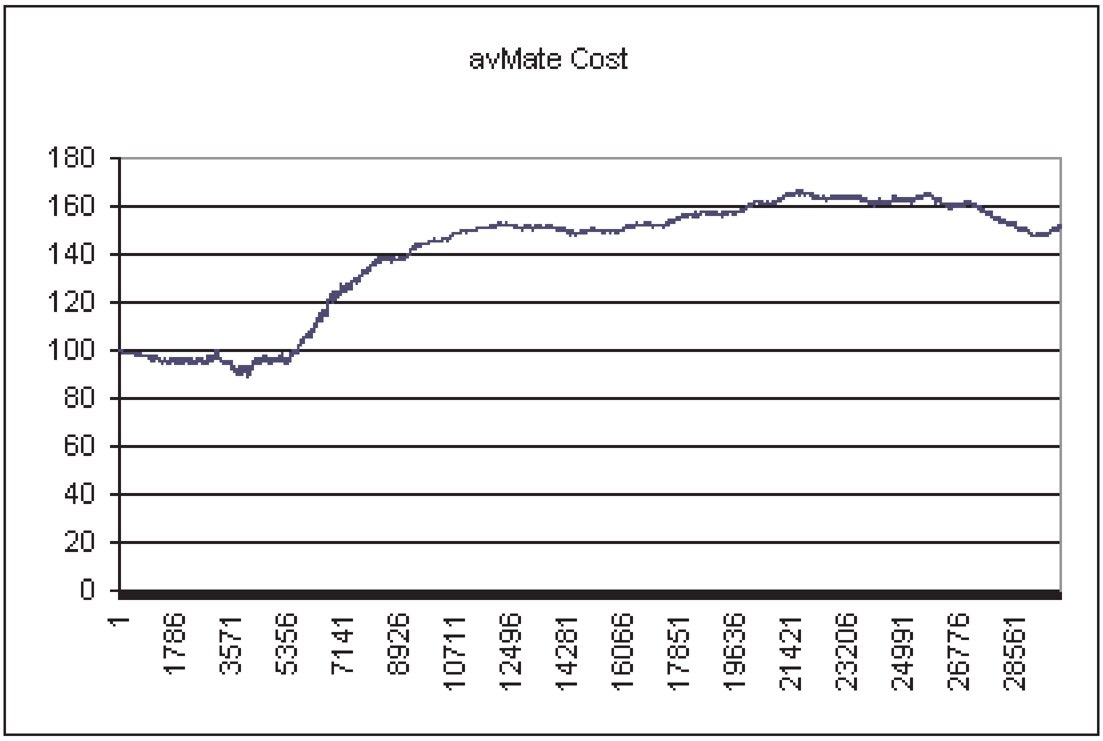


Chart of the average mating cost, for time steps 0 – 30000. The average mating cost increases sharply at the point where the population recovers from its initial decline, and then remains stable around 160.

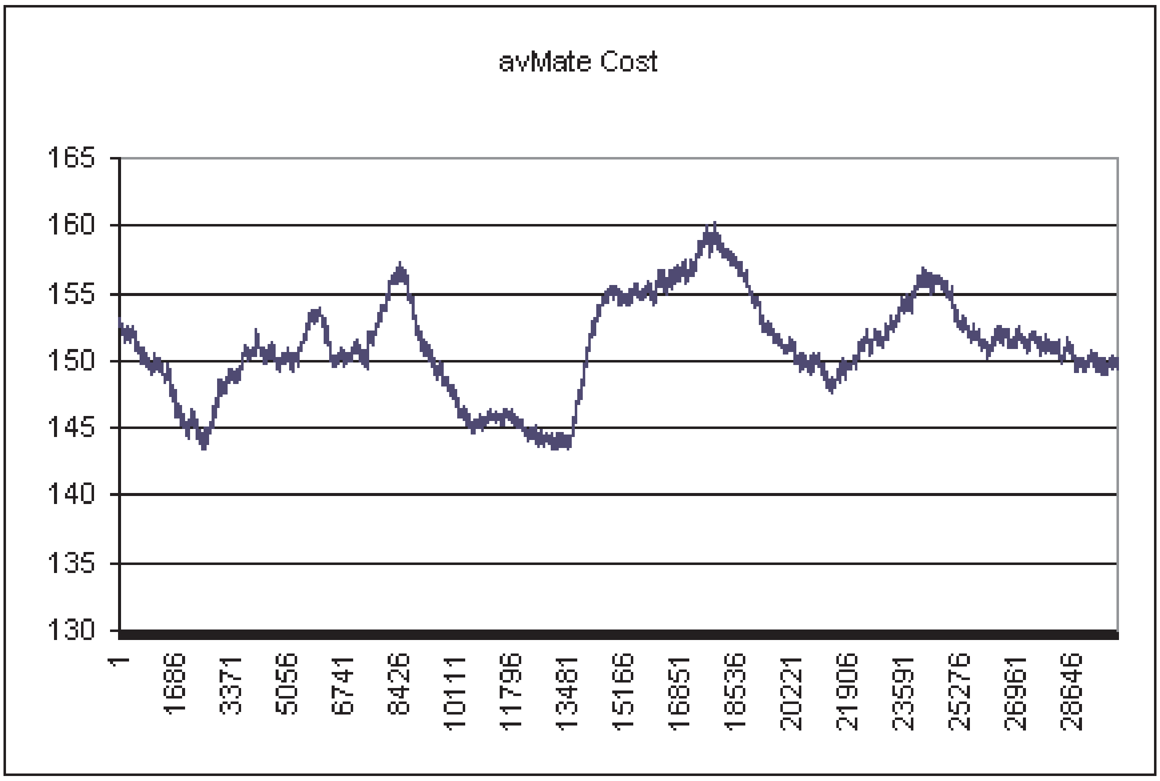


Chart of the average mating cost, for time steps 30001 – 60000. The average mating cost experiences fluctuations between 145 and 160.

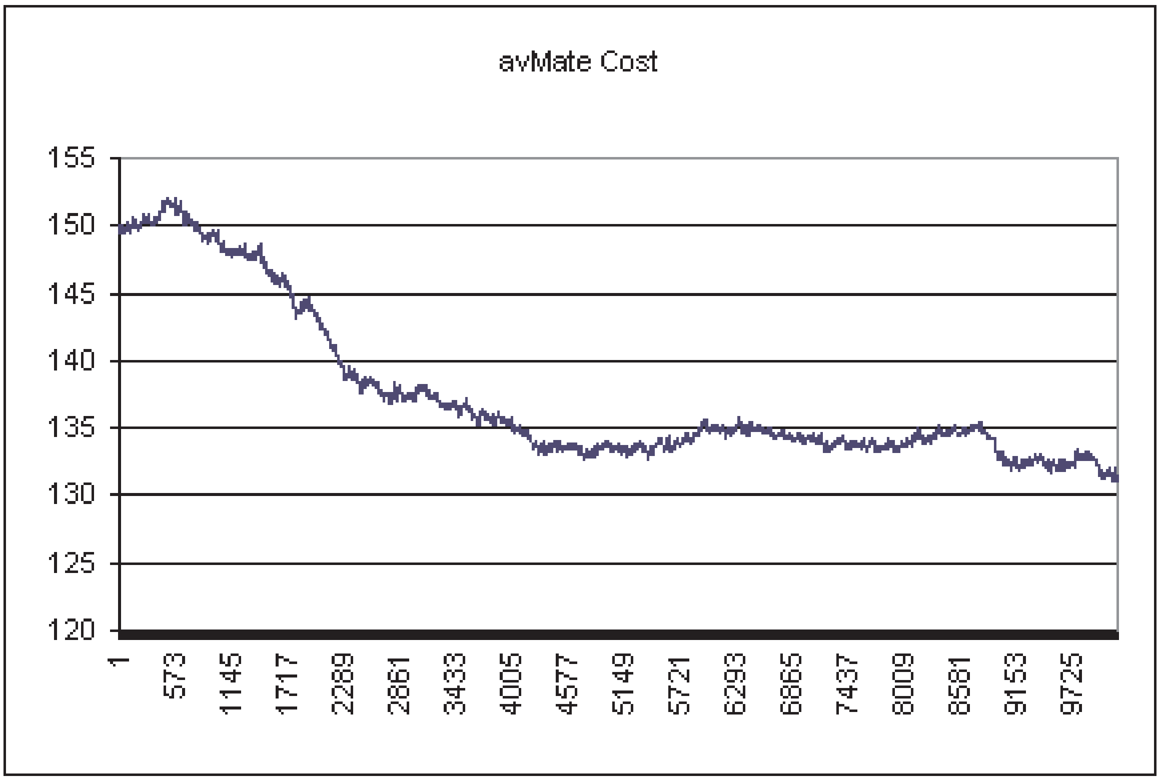


Chart of the average mating cost, for time steps 60001 – 70179. The average mating cost drops fairly quickly, but then evens out again at just over 130.

3.2.2 Interpretation

The degree to which this data differs from that of the first trial demonstrates the flexibility of the simulation. Many of the constants were modified, but only slightly. During the first 10,000 or so time steps the population is in risk of dying out. During this phase many different potential populations come into existence through mutation. We see herbivores, carnivores, cannibals and carrion eaters. In this trial, carrion eating turns out to be an effective way of surviving. Eating plants is of course also a very easy way to get energy, and as before the pure herbivore population is dominant.

In this trial the population reaches a point at about the 10,000th time step at which it stays for the duration of the trial. Changes do occur after this point, but they are gradual and small. After this point the animal population stays high and the plant population stays low. The number of corpses in the world also stays very low. Both plants and corpses are constantly being eaten. There is no cyclic behavior in this process. The number of plants and corpses being eaten stays fairly constant, which is what keeps both of these populations low. The corpse population is so low that the number of living animals nearly equals the number of total animals in the environment. Given only herbivores, and assuming that the rate of plant growth is such that the death rate of herbivores increases as the number of plants in the world drops, we would expect to see some periodic behavior as in the first trial. The herbivore-carrion eaters can switch back and forth between eating plants and corpses, so they are never lacking for a food source. Plants and corpses are constantly in low supply, but because both can be eaten it is enough to live on. Ironically, those animals that do suffer from lack of food improve the state of the world for the other animals when they die, because then their corpses can be eaten.

Energy levels remained low in spite of increased maximum energy levels. This implies that the energy was not available in the environment. In other words, the world functioned continuously at maximum capacity. Animals could not live long because they would starve. However, these individual deaths promoted the health and continuance of the overall population. The fittest carrion eaters were those that could survive long enough to eat their fellows after they had died. Staying alive would have been especially hard, given that animals had such high death points. However, this trait no doubt persisted in the population because it assured that dead animals were better food sources for everyone else – the amount of energy that a corpse is able to give to carrion eaters that eat it is equal to its death point. A higher death point means an earlier death, but it also means more food energy per corpse. In fact, given that mating costs and death points were high, carrion eating parents may have often become the food of their offspring, having died shortly after mating from starvation.

One trait that appears to have helped animals outlive each other is a small radius. Animals with larger radii lose energy quicker, and even more so when an animal has more than one eating behavior, as herbivore-carrion eaters do. This is particularly interesting because it is the opposite of what happened in the first trial. In trial 1, the average radius reached the maximum value because of the threat of predators and the need for predators to be big in order to eat. In this trial there are no predators, so having a large radius wastes energy. However, it also decreases the amount of energy that a carrion eater can gain on a single time step from a corpse it is eating – larger animals are assumed to take bigger "bites". In any case, evolution has favored the smaller animals over the larger. Animals with high speeds also seem to have been favored. This is interesting because there were no predators to run from. However, the animals were competing against each other for food, which we know from the low levels of plants and corpses was in small supply. Being fast was likely a way of getting to food before anyone else had a chance to eat it. The flip side of the coin is that larger movements cost more energy, and therefore catapult an animal towards starvation if it is unable to get food. This is likely why the average speed did not get as close to the maximum speed of 5 as it could. Rather, it settled slightly under 5, and fluctuated from time to time.

A high mating cost also helped animals to survive longer, though it also made mating more difficult. The amount of energy that an animal starts with at birth is the product of its starting energy and mating cost. As we see, the average value of the starting energy trait approached the maximum value of 0.9. From this point the only other way to increase the energy that offspring start off with is to evolve a higher mating cost. If the mating cost is too high, then an animal will never have enough energy to mate. If the mating cost is too low, then it will be easy to mate, but all offspring will be born weak with little energy, and therefore be likely to die before mating. In this trial the mating cost increased from its initial value. This makes mating more difficult, but this is not necessarily a bad thing. It assures that only those animals fit enough to reach the required energy are able to mate, and thus likely made the population stronger.

Another interesting occurrence in this trial is that the average learning rate increased from its initial value, unlike in the first trial where it became very low. Does this mean that Hebbian learning, as it is implemented in this simulation, is detrimental in a predator filled environment and insignificant otherwise? Or did the increased Hebbian learning rate help the animals of this trial to perform better and live longer? Hebbian learning is implemented at such a low level within the simulation that it would be very difficult if not impossible to determine exactly how it is influencing behavior, though this would be an interesting line of inquiry to pursue.

The last trait of note in the mutation rate, which begins to decrease after the population's decline near the start of the trial. That the mutation rate was decreasing implies that "change" was detrimental to survival. The dynamics of the population had apparently settled into a stable state, in which it attempted to remain. Presumably those animals with higher mutation rates gave birth to offspring whose mutations made it difficult for them to survive in the world that had become. Therefore selection favored those with lower mutation rates. It should of course be noted that the initial mutation rate of 1 used in both this and trial 1 is ridiculously high in comparison to real world systems. However, starting with a high mutation rate provides for quicker, and more interesting, results. In theory the same is possible by starting with a lower mutation rate, but it would certainly take longer for diversity to arise.

3.3 Trial 3: Shift in Dynamics

This trial was allowed to run for 37214 time steps, though it could have clearly run for much longer. The population became so large, that the speed at which the simulation ran was unbearably slow, especially since the population was still increaing at the point it was terminated. Although it would be very interesting to see how the trial would turn out given more time to run, the results that it provided within the given timeframe are quite interesting in and of themselves. We see in this trial an example of how the dynamics of the system can be established, last for a respectable amount of time, and then be replaced by an entirely different set of dynamics. At the point where this trial was terminated, the population shared some properties with that of trial 2 – a large population of herbivore-carrion eaters emerged – but before then the population had a more interesting mix of eating behaviors. The first half of the trial demonstrates periodic behavior similar but different to that pointed out in trial 1. Although periodic, the behavior is not that of Volterra-Lotka equations. The constants for this trial were set such that survival was much easier than in the previous two trials: more plants grew per growth step, energy costs were less and lifespans were longer. This seems to have opened up the simulation to a wider range of possibilities, which makes sense given that having more animals in the world leads to more possibilities, but the unfortunate side-effect of this is the drastic speed decrease. Given more time and/or faster machines, this trial could be carried out for a longer period of time, which could lead to results even more interesting than those provided in such a short timeframe.

3.3.1 Data

interactionDistance = 15	minMatingTime = 0	growthTimeStep = 5
speciesEqualityConstant = 0.15	maxMatingTime = 3	plantsPerGrowth = 110
minStartEn = 0.1	sStartEnergy = 0.6	plantNutrition = 65
maxStartEn = 0.9	sSight = 0.0	binGridSizeX = 2
minSight = 15	sHerbivore = 0.6	binGridSizeY = 3
maxSight = 70	sCarnivore = 0.35	binWidth = 100
minManeuverability = 0.1	sCannibal = 0.35	binHeight = 100
maxManeuverability = 3.0	sCarrion = 0.49	initialPopulation = 200
maneuverabilityRestriction = 0.1	sManeuverability = 0.0	initialPlants = 300
minMoveSpeed = 1.0	sSpeed = 0.0	netArchitecture = [26,26,26,10,2]
maxMoveSpeed = 5.0	sLifespan = 0.4	maxFloatMutation = 0.35
minMove = 0.2	sMutation = 1.0	maxIntMutation = 4
moveCostMultiple = 0.05	sRadius = 0.0	initialEnergy = 200
minMaturity = 30	sMateCost = 0.3	metabolicRate = 0.02
maxMaturity = 70	sLearning = 0.1	decay = 0.3
minLifespan = 500	sMaturity = 0.4	seed1 = 17035
maxLifespan = 1500	sMateTime = 0.5	seed2 = 20573
minMutationRate = 0.1	sDummy = 0.5	seed3 = 30189
maxMutationRate = 1.0	sMateRecovery = 0.4	seed4 = 57143
minRadius = 5	deathEnergy = 0.7	seed5 = 16529
maxRadius = 20	energyRestriction = 25	seed6 = 24937
minMatingCost = 50	predationMultiplier = 10	seed7 = 27717
maxMatingCost = 400	predationConstant = 340	seed8 = 82144
minMatingRecovery = 40	carrionMultiplier = 19	seed9 = 56724
maxMatingRecovery = 80	cannibalGracePeriod = 40	seed10 = 33221

Constants for trial 3

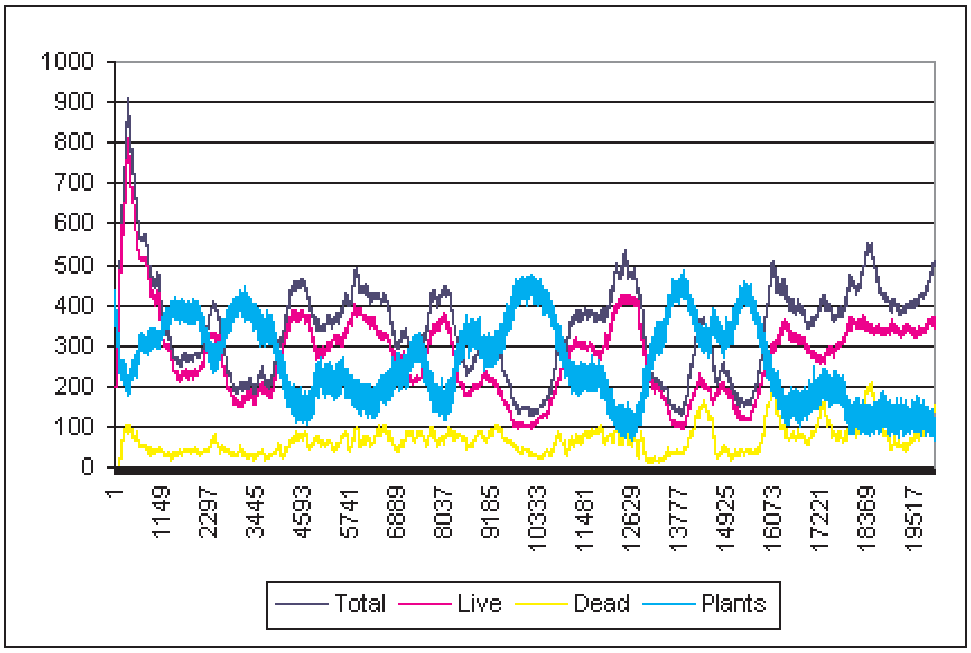


Chart of the total animals, live animals and dead animals, along with the number of plants, for time steps 0 – 20000. From the start, periodic behavior of both the animal and plant populations is clear to see. Although periodic, this behavior is not typical Volterra-Lotka behavior, but it does resemble the behavior initially seen in trial 1. Every peak in the plot of the plant population seems to correspond to a valley in the plot of the living animal population. The fluctuations of the two plots seem to be equal in opposite directions. Around time step 18000 the wild fluctuations cease.

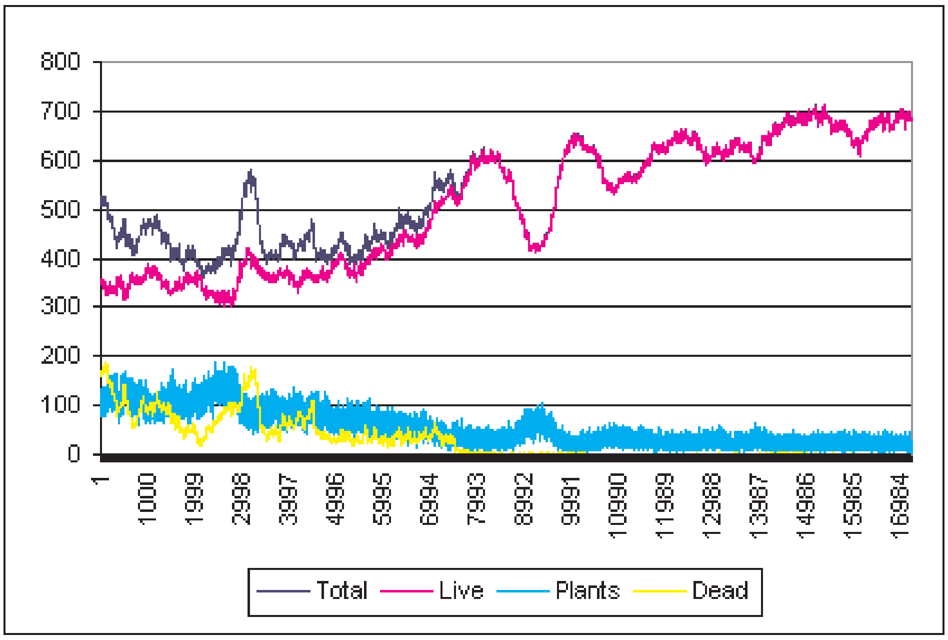


Chart of the total animals, live animals and dead animals, along with the number of plants, for time steps 20001 – 37214. This data set is surprisingly different from the previous set. The plant population shrinks as the population of live animals grows ever larger. This is deceptive, because plants are only counted after animals have acted. Therefore the high population of animals may have saturated the environment such that new plants are eaten before counted. The number of dead animals in the world approaches 0. Though the behavior of the plant and animal populations in this timeframe is very different from that of the first timeframe, the two plots still influence one another, as evidenced by a sharp drop in the animal population around time step 9000 that corresponds to a slight peak of the plant population at the same point.

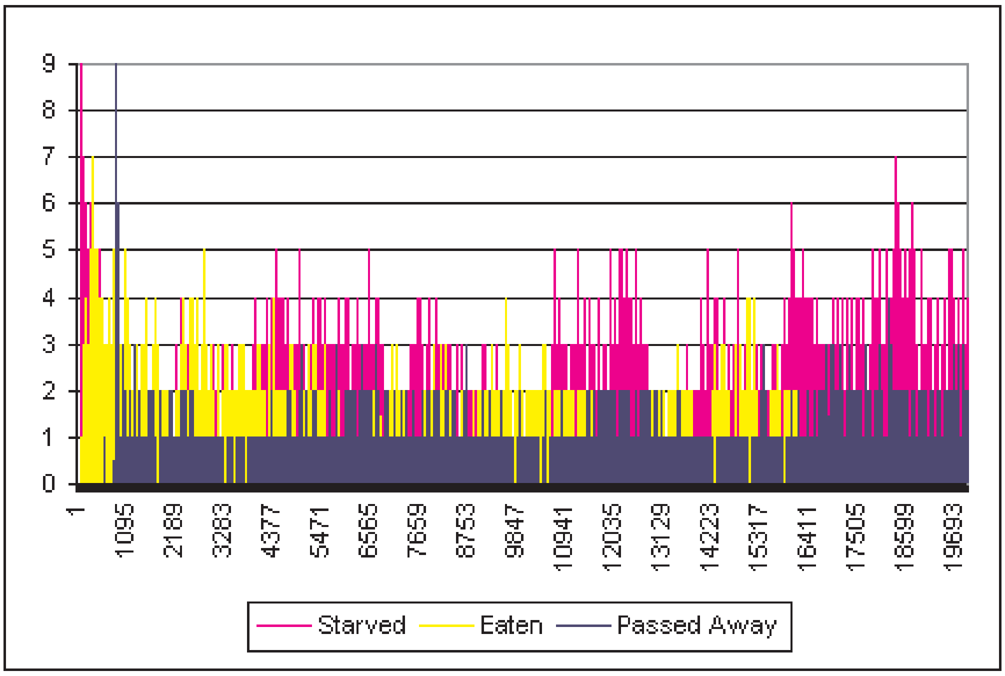


Chart of the number of animals that starved, died from being eaten, or passed away from old age, for time steps 0 – 20000. At the start of the trial, death by starvation is fairly common, as is death from being eaten. Except for the expected peak at the start of the trial, passing away from old age occurs less than the other two forms of death. Near the end of this timeframe, at the point where the plant population stops fluctuating and remains small, deaths from starvation increase. Death by old age also occurs with greater frequency. Directly before this point, death by being eaten decreases.

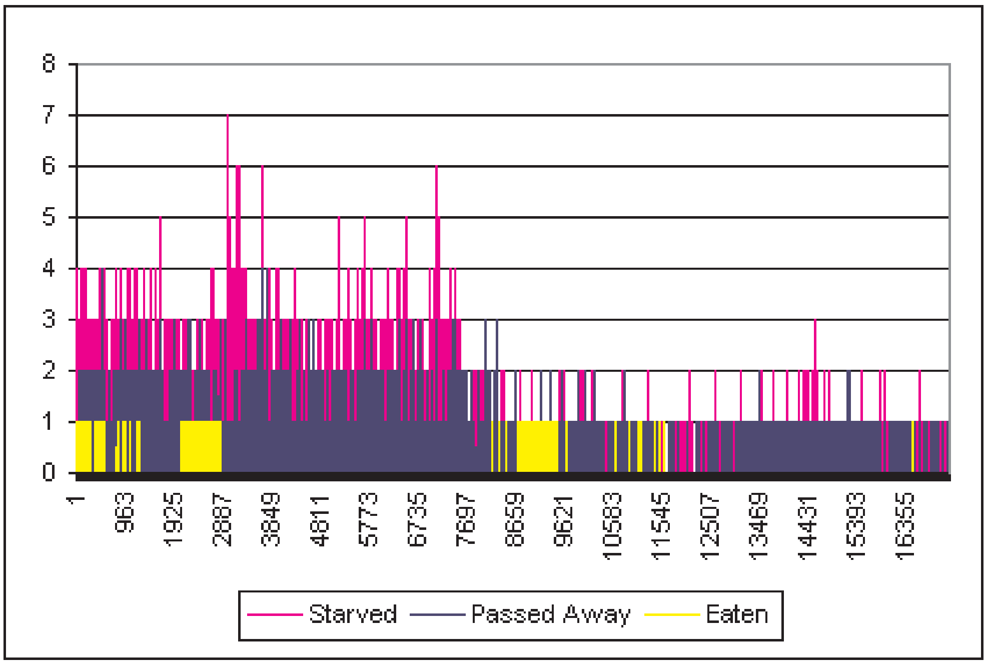


Chart of the number of animals that starved, died from being eaten, or passed away from old age, for time steps 20001 – 37214. Death by starvation remains common for the first half of this timeframe, as does death from old age, though to a lesser extent. Death from being eaten now occurs seldom, though in concentrated blocks. It is interesting that deaths by starvation become infrequent during the second half of this timeframe, given that the plant population remains especially low. This indicates a shift in how the animals are obtaining energy.

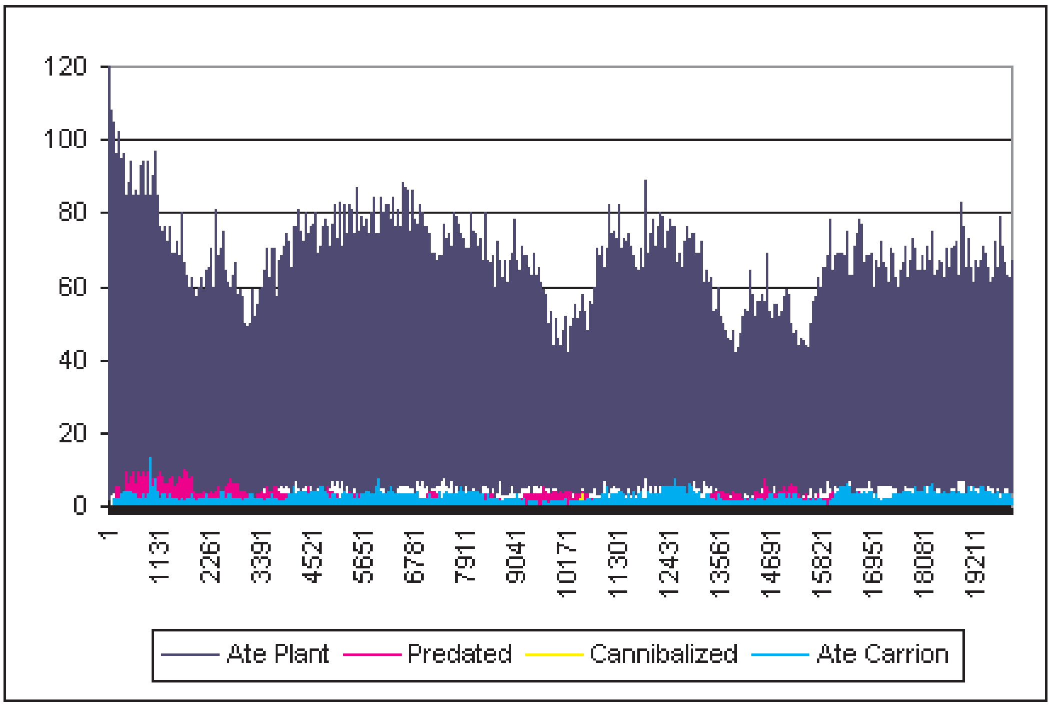


Chart of the number of animals that ate plants, ate an animal of another species (Predated), cannibalized another animal, or ate carrion, for time steps 0 – 20000. Plant eating remains dominant, as usual, but predation and carrion eating are present throughout the timeframe. Cannibalism occurs seldom.

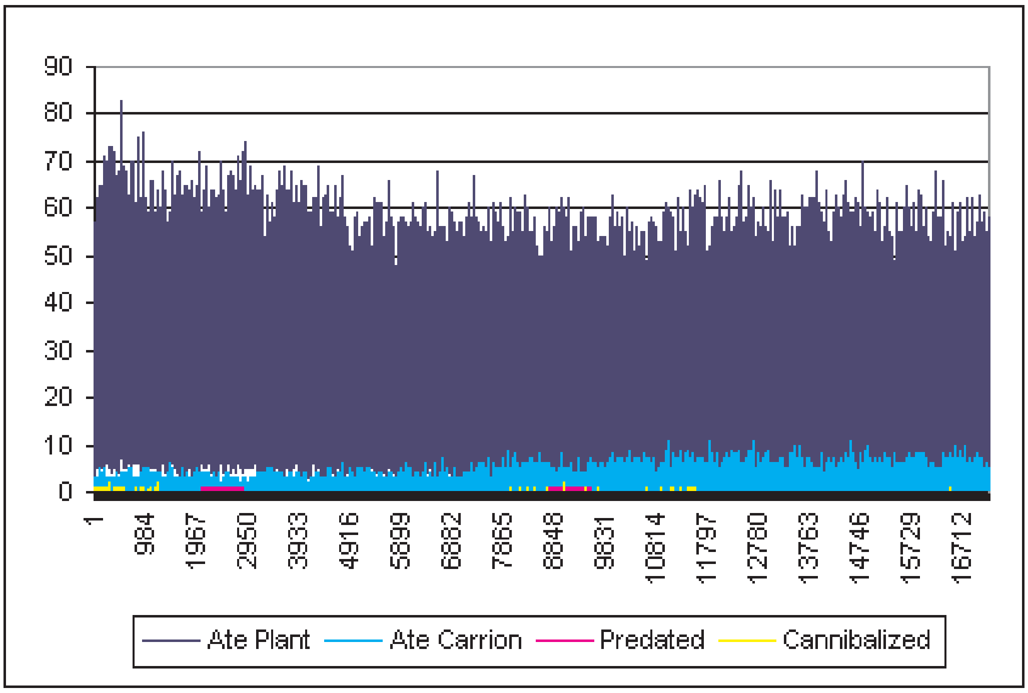


Chart of the number of animals that ate plants, ate an animal of another species (Predated), cannibalized another animal, or ate carrion, for time steps 20001 – 37214. Plant eating remains popular, indicating that nearly all plants are being eaten before being counted (otherwise the data would indicate that more plants were being eaten than were available to eat). Carrion eating greatly increases in popularity, which likely accounts for the decrease in deaths by starvation despite the low plant population. Cannibalism remains rare, and predation has become just as rare.

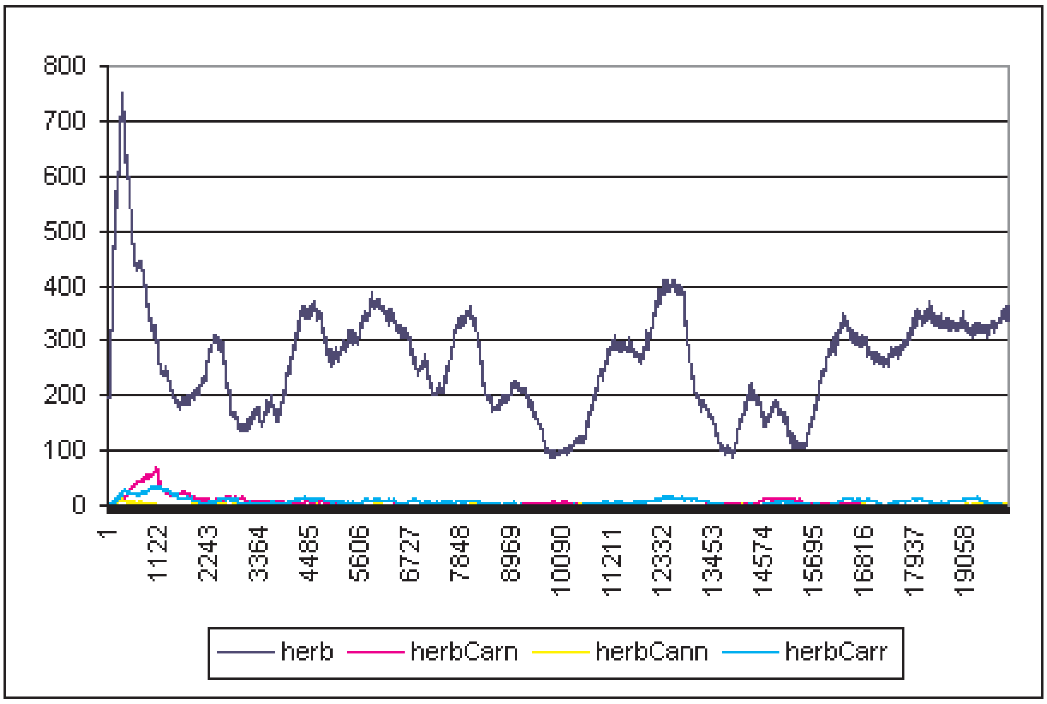


Chart of the number of pure herbivores, herbivore-carnivores, herbivore-cannibals and herbivore-carrion eaters, for time steps 0 – 20000. As expected given the behavior of the animals, the herbivore population is the biggest. Its fluctuations correspond to the fluctuations of the overall population shown earlier. The population of the other types of animals remains quite small throughout this timeframe.

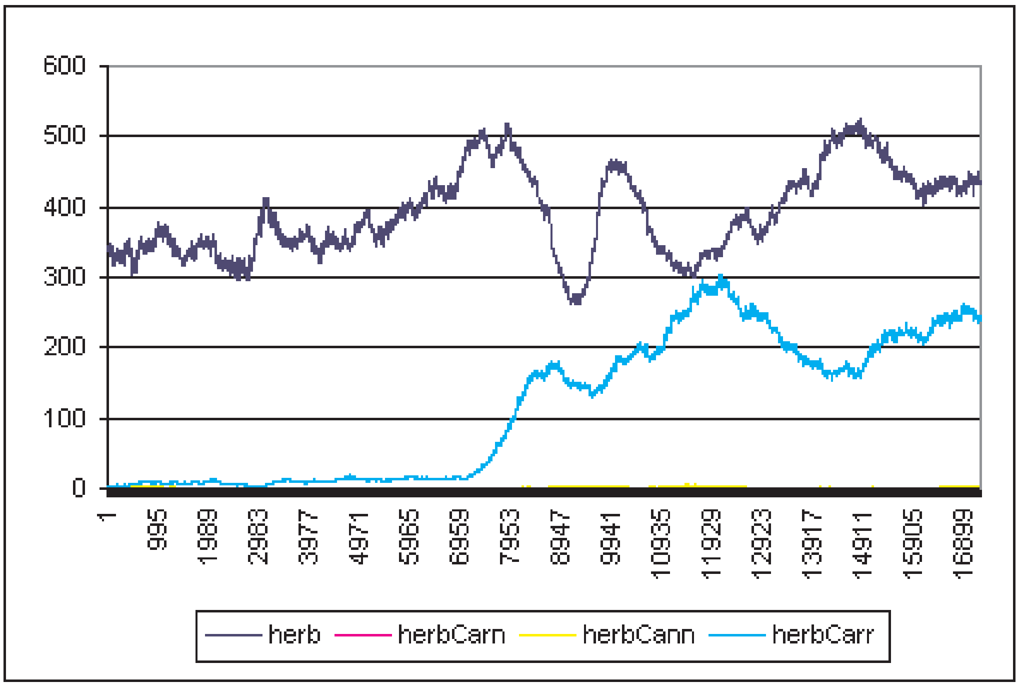


Chart of the number of pure herbivores, herbivore-carnivores, herbivore-cannibals and herbivore-carrion eaters, for time steps 20001 – 37214. The herbivore-carnivore and herbivore-cannibal populations remain small, and the pure herbivore population remains large, but the herbivore-carrion eater population experiences a drastic increase around the midpoint of this timeframe. It is around this point that deaths by starvation decrease and carrion eating increases. Note also that the sudden drop in the overall living animal population seen earlier around time step 9000 is caused entirely by a drop in the pure herbivore population. The herbivore-carrion eater population is increasing at this point.

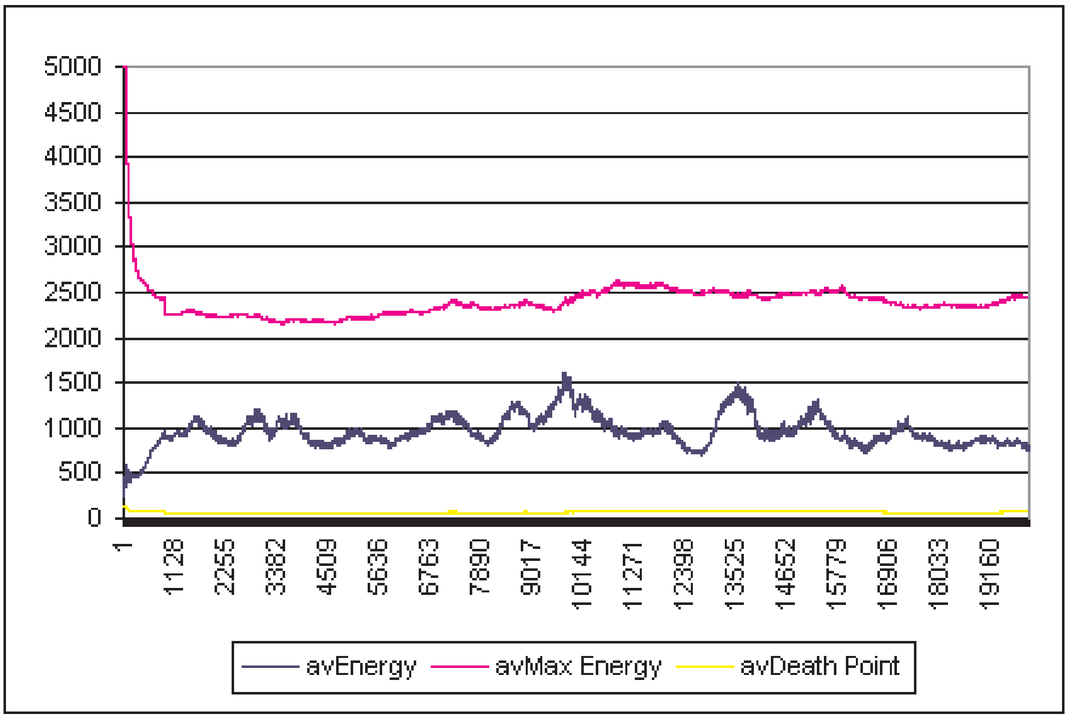


Chart of average energy levels, average maximum energy levels, and average death points, for time steps 0 – 20000. After the initial population, whose maximum energy level is very high, dies out, the average maximum energy stays fairly stable around 2500. The average death point stays low, and the average energy level fluctuates comfortably between the two.

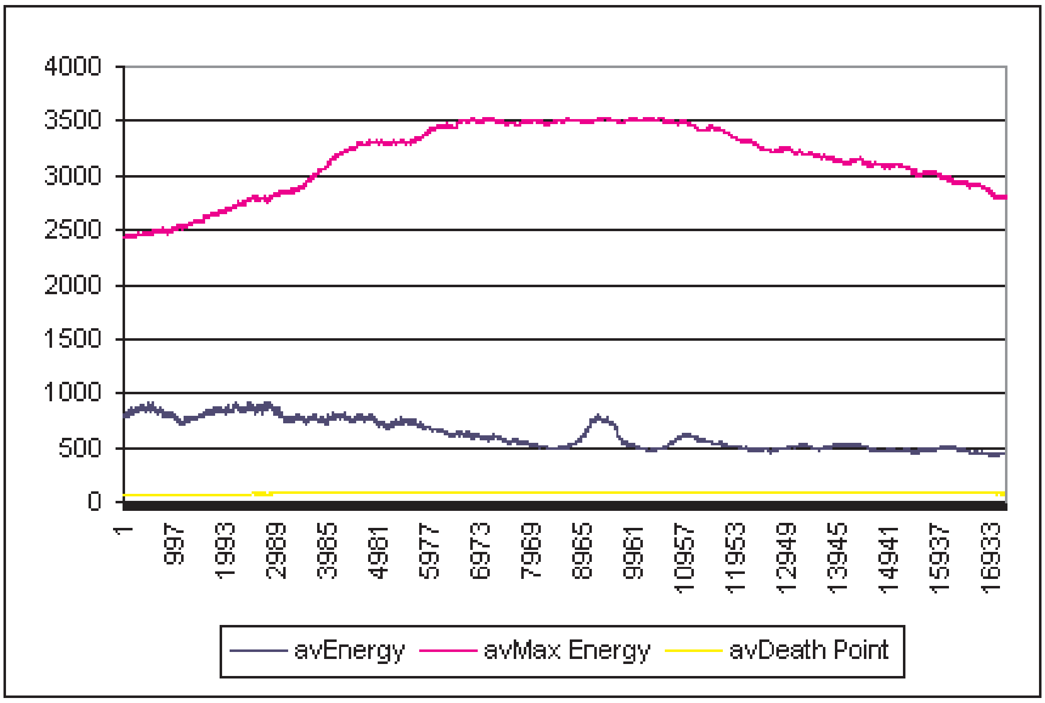


Chart of average energy levels, average maximum energy levels, and average death points, for time steps 20001 – 37214. The average maximum energy level increases by about 1000 and stays around 3500 throughout the middle range of this timeframe. In spite of increased maximum energy levels, the average energy level decreases steadily throughout this timeframe, likely due to lack of food from overpopulation.

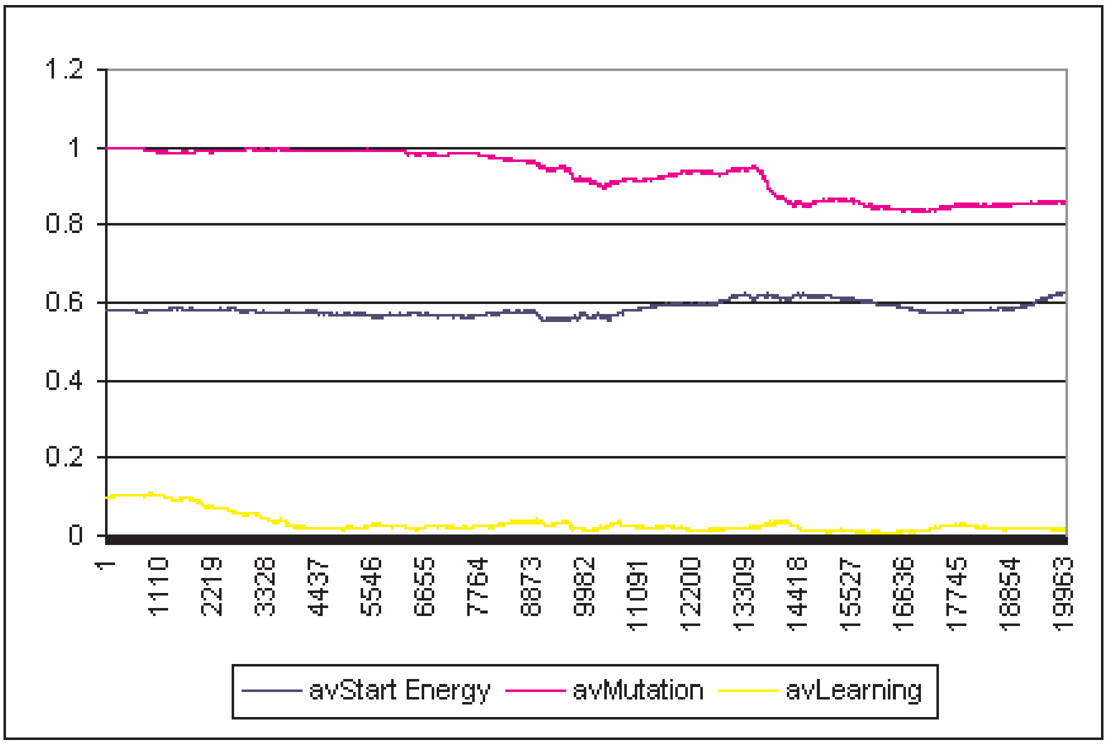


Chart of average starting energy levels, average mutation rates, and average learning rates, for time steps 0 – 20000. As is common, the average mutation rate decreases from its maximum of 1. The average learning rate decreases to almost 0 as it did in trial 1. For the moment, the average starting energy remains stable.

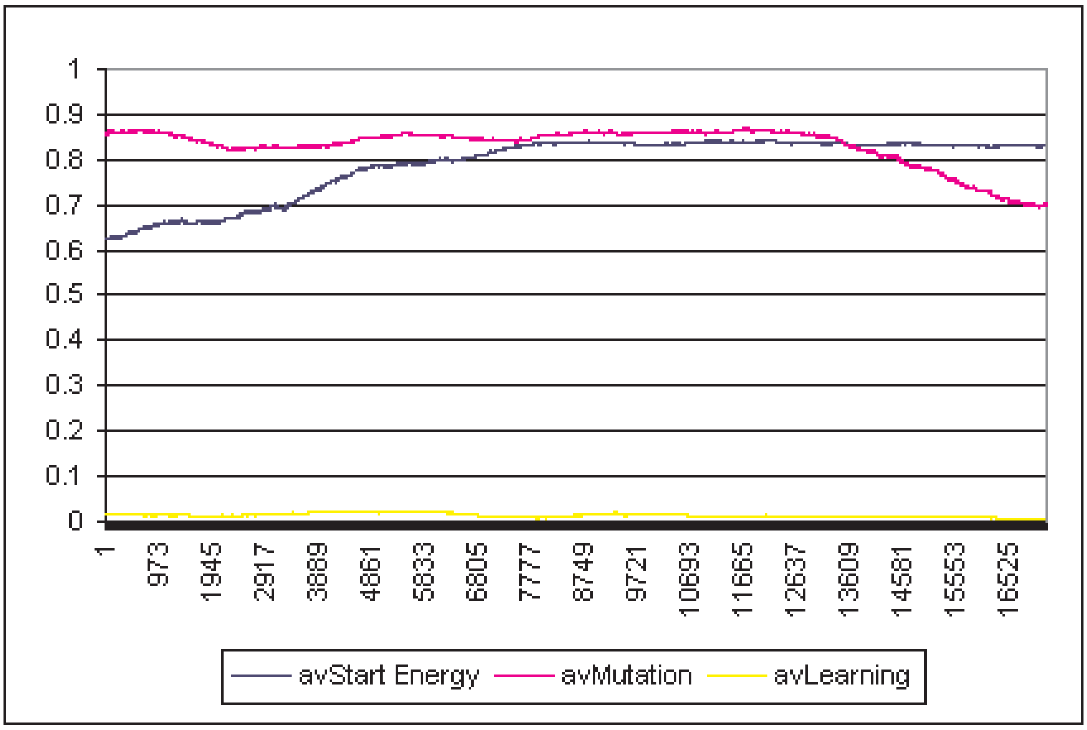


Chart of average starting energy levels, average mutation rates, and average learning rates, for time steps 20001 – 37214. The average mutation rate continues to decrease and the average learning rate remains close to 0, but the average starting energy increases greatly, coming close to its maximum value of 0.9.

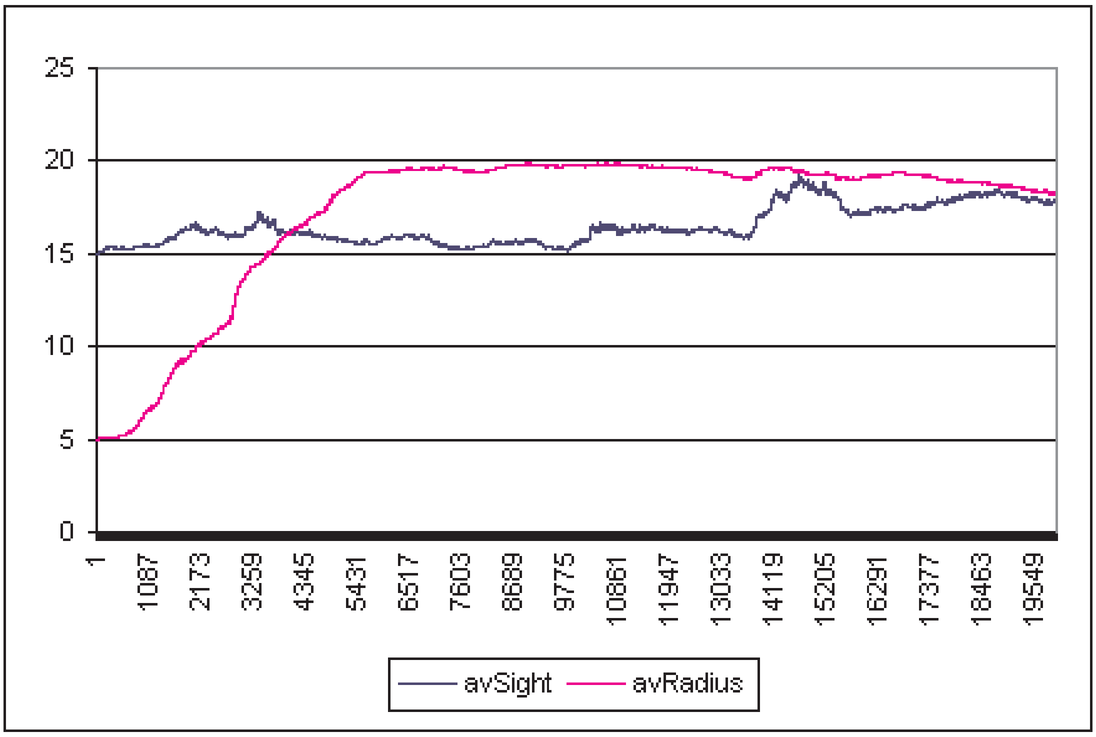


Chart of average sight ranges and average radii, for time steps 0 – 20000. The average radius quickly approaches its maximum value of 20, and remains there throughout this timeframe. The average sight range does not increase in response to this, meaning that a significant portion of the animals are larger than they can see – an amusing situation.

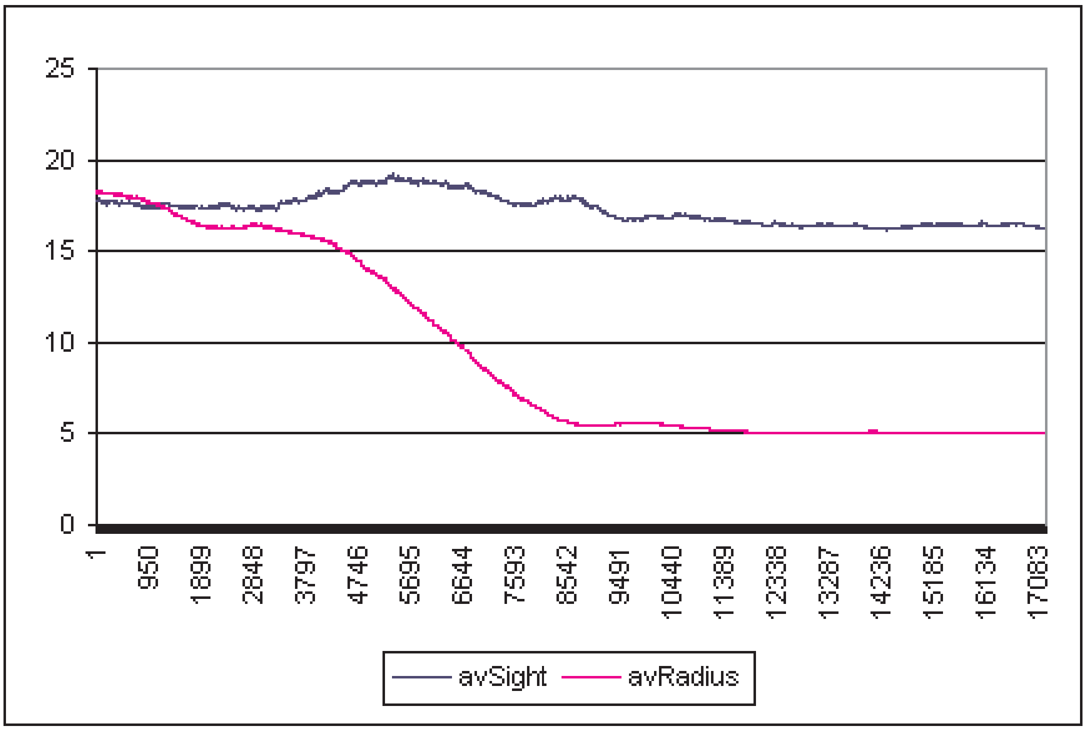


Chart of average sight ranges and average radii, for time steps 20001 – 37214. The average sight range remains fairly steady, but having a large radius proves to be detrimental in the second half of this trial, and thus plummets back to the minimum value of 5. This drop coincides with the emergence of a large herbivore-carrion eater population.

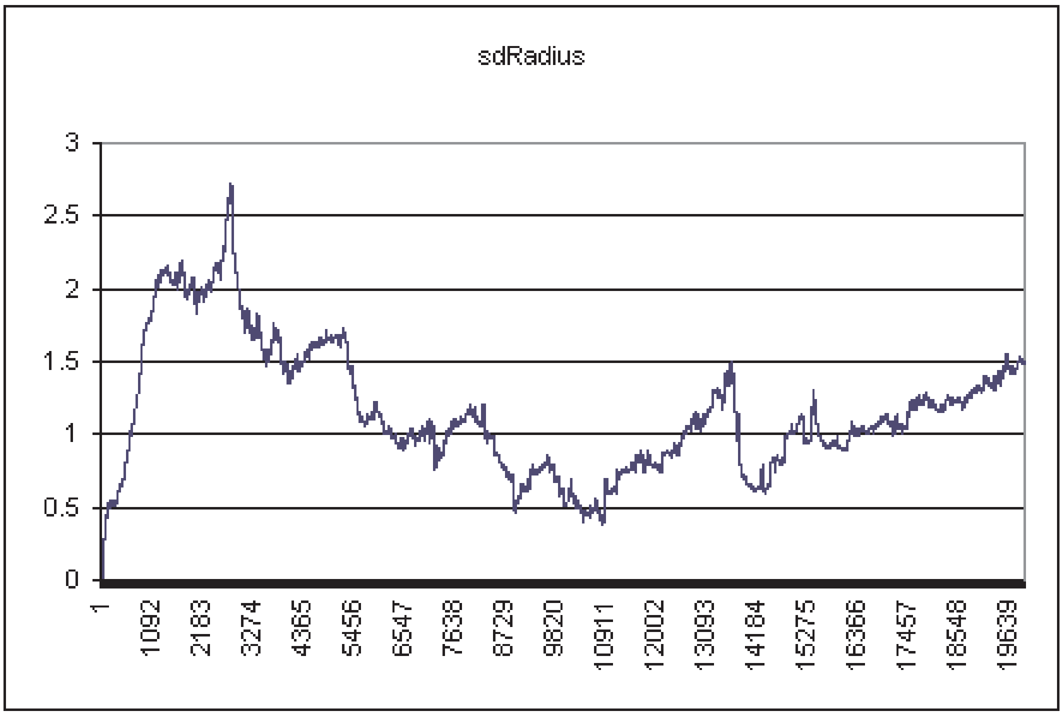


Chart of the standard deviations between animal radii, for time steps 0 – 20000. The offspring of the initial population shows a great deal of variation, which results in the high standard deviation at the beginning of the trial. The standard deviation between radii decreases as the average radius increases, therefore the majority of the animals are large at this point. This seems to be the common reaction to the presence of predators. At the end of the timeframe the standard deviation between radii increases again.

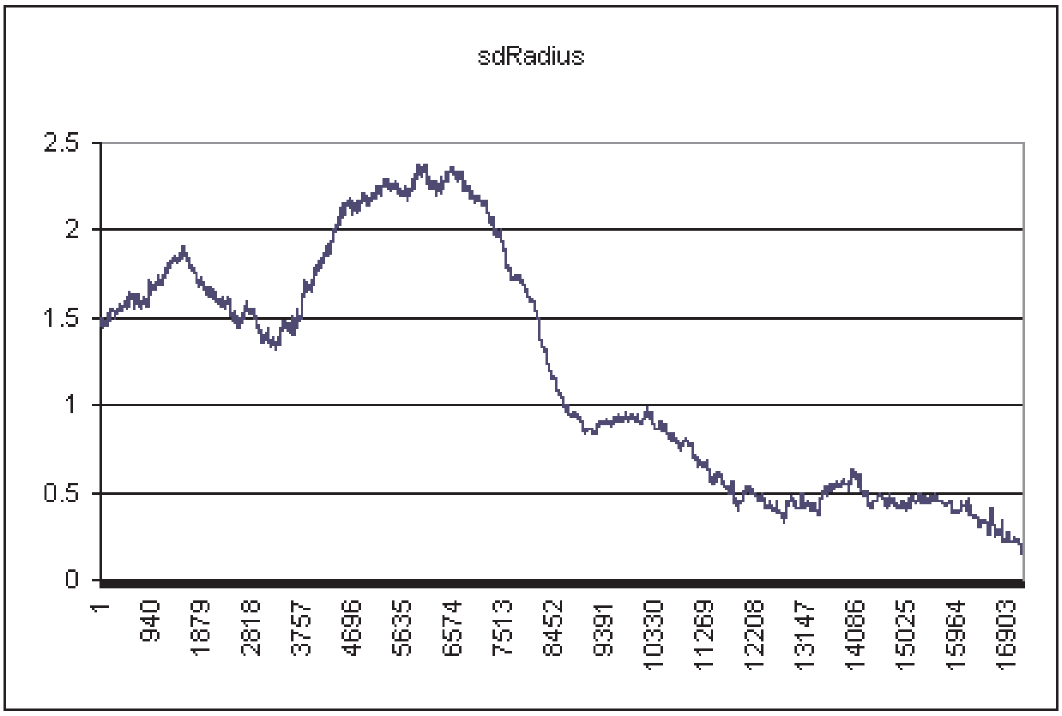


Chart of the standard deviations between animal radii, for time steps 20001 – 37214. Variation between radii continues to increase, likely because of the lack of preadators. However, after peaking around time step 6500, the variation greatly decreases, until the standard deviation is very close to 0. This decrease in variation happens at the same time that the average radius shrinks to its minimum, meaning that the entire population is small at this point.

3.3.2 Interpretation

Before the emergence of carrion eating as a popular eating behavior, there is a periodic relationship between the plants and the living animals, nearly all of which can eat plants. We have already seen that plants seem to be the primary food source of all successful animal populations, even if they do possess another eating behavior. Thus it makes sense that the animal and plant populations would respond to each other, though the manner in which every peak of one population corresponds to a valley of the other is confusing. This seems to indicate that both populations are in fact responding simultaneously to a third and unknown factor. Either that, or the time lag between a change in one population and the other population's response is extremely small, which seems unlikely. An in depth analysis of local interactions could perhaps provide more insight into this conundrum.

The periodic behavior stops as the population becomes exceedingly large. The large animal population devastates the plant population. Had not plant growth been modeled in such a simple manner, such a large population could likely drive the plants to extinction, but as it is the animal population manages to survive while keeping the plant population very low. This scarcity of food leads to many animals dying of starvation, until carrion eating gains a strong pressence within the population. Such deaths happen seldom at this point, because the flourishing carrion eating population does not need to rely exclusively on plants. This situation is very similar to that of trial 2.

Throughout the first half of the trial, the herbivore-carnivore and herbivore-carrion eater populations, though both small, are nearly equal. It would have been interesting to see a trial in which three different populations had a significant pressence, but the requirements of a world based on predation and those of a world based on carrion eating do not appear to be compatible. Predators can only be successful when they are large, which causes the entire population to become very large. The average radius approaches the maximum during the first half of the simulation, which demonstrates the simulation's orientation around the, admittedly small, herbivore-carnivore population. However, once each animal has a radius approximately equal to the maximum radius, the amount of energy that a predator is capable of gaining from eating an animal is smaller. Because of this, herbivore-carnivores can likely only sustain themselves if they can eat plants in addition to other animals. This is a problem when the animal population becomes so large that the plant population remains close to 0. This is likely why the predators died out. With no predators to worry about, having a large radius lost its purpose, so the average began to decrease. The large herbivore-carrion eater population does not emerge until after the average radius has become significantly small. This could be a mere coincidence, but it also occurs in trial 2. It seems that a population of carrion eaters is only inclined to evolve from a population whose members have small radii. If this is true, it makes the simultaneous pressence of both large predator and large carrion eater populations unlikely, but not impossible. Attempting to create such a population through reconfiguration of the constants could be an exciting endeavor.

Yet another way that the behavior of the trial changes between the first and second halves is the increase in average maximum energy level. Maximum energy level and death point are both directly related to starting energy, which also increases in the second half of the trial. A large starting energy can be detrimental, because it also increases the death point. This is likely part of the reason that this trait remained close to the starting value for the first half of the trial. In a predator oriented world, large size causes one to lose energy too quickly for a high starting energy to be able to offset the disadvantage of a high death point. For predators especially, the uncertainty about when the next meal will come makes a high death point particularly detrimental. Therefore it is only later, when the predators have mostly died out, that the average starting energy increases, and with it the average maximum energy and average death point. This occurs before the emergence of the herbivore-carrion eater population, and may be yet another precondition for such emergence: the same occurred in trial 2. This makes sense, because a high death point puts more corpses in the world, which provides a source of food for a fledgling carrion eater population.

However, it is worth note that that average maximum energy starts to decrease again at the end of the trial. The world was likely on the verge of yet another fundamental change, and it is unfortunate that the results of the change were not recorded. Would yet another species have emerged? Would a population with three eating behaviors have developed? Or would the population have gone extinct? Maybe it would have simply become a homogeneous herbivore population again. Though the results are not presented in this document, that can be easily obtained through recreation of this experiment . . . and some patience.

3.4 Other Trials

The three trials presented above were three of the most interesting trials, but a great many more trials were also performed in order to get the results above. Through trial and error, and a great deal of random choices, constants were adjusted in hopes of generating interesting data sets. Often a change in a parameter would cause the opposite of the expected result, and sometimes simply changing the seeds for the random number generator resulted in a huge qualitative difference in overall behavior.

Of the trials not presented, many were populations that quickly went extinct in a rather uninteresting fashion. Sometimes the initial herbivore population would die out before any new eating behaviors arose, and at other times the emergence of a new eating behavior seemed to be the direct cause of the population's extinction. There were also trials in which the animals flourished, but otherwise did nothing of interest. There were several trials in which a herbivore population grew at a seemingly exponential rate, the resulting slowdown of which required that these trials be terminated. There was even one peculiar trial in which a very small population composed exclusively of herbivore-carnivores lived for over 100,000 time steps.

Therefore, this simulation is capable of creating very uninteresting data in addition to the more interesting trials presented above. Some of the populations not presented went extinct too quickly to be of interest, or grew too quickly to make continued execution of the simulation practical. This hints at some untapped potential for the simulation, which can only be unlocked with either faster computers, or great patience.

3.5 Conclusions and Future Possibilities

Using the functional programming language Haskell, an Artificial Life simulation has been modeled. Modeling with Haskell has several advantages, yet also poses some interesting problems. First there was the issue of modeling randomness in a deterministic environment using a purely functional language. The linear congruential method of random number generation can be implemented in any programming language, but the availability of streams greatly simplifies implementation in Haskell. Then there was the problem of creating abstract data types suitable to Haskell. Such data types are available in the Haskell community, but when deciding which to use, one must take special care to consider how functional languages differ from the imperative languages that most of us are so used to.

Another step in the modeling process involved two important contributions from the field of Artificial Intelligence, which have been liberally applied in the field of Artificial Life: genetic algorithms and neural networks. Being based on biology themselves, it is only natural that these tools should be applied to Artificial Life, which attempts to both mimic and surpass the biological world. However, it is necessary to modify these tools to fit the needs of Artificial Life. The field of Artificial Intelligence burdened both of these tools with some baggage that must be stripped away before one can properly apply them to Artificial Life.

Typical genetic algorithms are very synthetic in their use of fitness functions. A fitness function provides a computer scientist with a useful heuristic for solving a well-defined problem. While effective, such a tool is not "natural". It is a highly objective measure of fitness, based solely on the aims of the programmer. A programmer's use of a fitness function is like a breeder of horses choosing to place a stallion and mare together in the same corral. It gives rise to offspring with particular desirable traits, but the choice of which traits are "desirable" is biased. By removing the fitness function and placing the animals in a living world, the choice of what aim the population strives towards is taken out of the hands of the programmer. It is of course true that the programmer controls the simulation, and that some traits may be defined, such that they are very likely to be beneficial, but these traits are not specifically designated by the programmer as being able to determine who lives and mates. That decision is up to the animals themselves, and the choices they make can often surprise even their creator.

Neural networks are also commonly used in a synthetic manner. The first algorithms developed for neural networks were supervised learning methods, which train neural networks to respond to inputs such that they can solve specific classes of problems. The goal is designated at the start, and the weights of the neural network are adjusted to help the network reach this goal. It is not uncommon for the synaptic weights of a neural network to become fixed once the network comes close enough to the goal set for it by its designer. It becomes a static entity. Even when unsupervised learning methods such as Hebbian learning are used, the "optimal" set of synaptic weights attained from the learning process often becomes fixed. Learning is switched off. When using neural networks to represent living organisms, learning should never be switched off. Learning continues throughout the lifetime of an animal, and never reaches a final goal, but rather continues to strive towards the goal, whatever it may be.

However, such a situation is extremely hard to model. How does one allow for infinite possibilities in the confines of a discrete machine? In the course of constructing this model, many decisions were made for the sake of performance or tractability. Some aspects are based on the real world, but many are not. Some features were added because their lack gave rise to undesired situations. For example, an earlier version of the simulation imposed no penalty for possessing multiple eating behaviors. This nearly always resulted in a homogeneous population of herbivore-carnivore-cannibal-carrion eaters. Other features were added for similar reasons. In this sense, the programmer does behave somewhat like a horse breeder, but the modeling process requires that decisions be made. It is inevitable that some of these choices impose restrictions on the world. The name of the field is, after all, *Artificial* Life. Still, organisms in the real world must also deal with restrictions. The physics of the universe is a set of constraints that all life faces, and life has risen to the challenge.

The animals in the simulation have also risen to the challenge. The simulation has shown itself to be capable of producing several different types of worlds with various types of organisms. That each of the trials had at least two distinct types of animals shows that either sympatric or parapatric speciation occurred. Different populations emerged from a single starting group. This diversity depends on the values of the 75 constants controlling the simulation, and given the range of values that these constants can take on, there remain many untried possibilities – we have only scratched the surface. Further analysis could be done to determine how the simulation behaves with respect to certain constants, which is something that we purposefully refrained from doing. In this way one could demonstrate which constants are afflicted by the Butterfly Effect and which are not. It would also be interesting to see how a trial's results change when nothing but the random number seeds are changed. Would the behavior be significantly different?

However, instead of simply testing every possible combination of starting values in order to obtain more and more interesting results, we could go back to the modeling phase and rethink some of the assumptions made within the simulation. That predators must be larger than prey in order to eat them has already been identified as a problematic assumption. This was done to vary the amount of energy that a predator got from eating, which is the size difference between predator and prey. If this restriction were removed, then energy transfer could be based solely on the size of the predator, or perhaps on some other trait. In any case, it would certainly have a strong impact on the output of the simulation.

Another quirk of the simulation is the action order. Mating precedes eating plants, precedes eating other animals. This simplifying assumption influences animal behavior in an artificial manner. In order to make these decisions more "natural", various decision schemes are possible. Simplest would be to have animals make a random choice when faced with multiple decisions. This option at least allows for more variation in behavior than a static action hierarchy. For choosing between multiple food sources, animals could be programmed to choose the source providing the largest potential energy gain. The decision to mate could be made more interesting by modeling estrus cycles, and thus making females receptive to mating only at certain times. Modeling this would also distinguish males and females slightly; a distinction that is currently lacking from the simulation. The most interesting, and most computationally complex, method for animals to make decisions would be for them to have an additional neural network for determining action. This would in effect take the decision out of the hands of the modeler and hopefully lead to more "natural" behavior.

Besides modifying the model, we could also improve our data collection and analysis methods. The preceding analysis focused primarily on population counts and the traits and properties of the animals (in other words, data members of the `Traits` and `Animal` instances). No analysis of the neural networks or the two types of chromosomes that produce them, weights and time delays, was performed. The `Body` instance of the animals was completely ignored. This was done because the complexity of the neural networks is difficult to penetrate. It would be nearly impossible to infer the effect of a particular synaptic weight on the behavior of an animal. However, this does not mean that the mysteries of animal behavior within the simulation are impenetrable.

Animal behavior within the simulation can be studied in a manner similar to the laboratory experiments biologists do. Animals in a lab environment are easier to study because it is a controlled environment. Biologists can present to animals various stimuli and observe the animals' reactions. We can do something similar with the simulation by taking the definitions of animals in terms of their chromosomes and recreating these animals in a controlled environment. Individual predators could be observed to see if they chase prey. Small populations could be observed for evidence of schooling behavior. Eating preferences could also be examined – although animals cannot choose what to eat when faced with several options, they can choose what food to move towards.

Recreating animals within a lab environment would require some slight modifications to the simulation. In its current form, information about weight and time delay chromosomes is not part of the output on each time step. This is because the amount of time required to output this additional information on every time step greatly reduces the speed at which the simulation runs. However, for the sake of studying animals in a lab environment it would be sufficient to only output this information every 1000 time steps, or perhaps even less often. Using such data to create lab animals, one could even mix animals from different trial runs to see how generalized their behavior had become: do they only respond to the specific animals they originally lived with, or do they react to animals from other trials in the same way?

Beyond further analysis of the results, the simulation is also useful as a teaching aid. Evolution by natural selection is clearly present in each trial of the simulation. In trial 1 the presence of predators naturally selects for animals that have large radii. In trial 2 nature selects for animals with small radii. In both trials animals with higher maximum energy levels are selected for. Trial 3 demonstrates a shift in natural selection as a result of the population changing: the previously beneficial trait of large radius becomes detrimental in the second half of the trial. These changes are distinct shifts in the genetic makeup of the population.

If an interface were to be developed that allowed for easy modification of the simulation parameters, then this simulation could be used as a hands on means of teaching students about evolution by natural selection. These phenomena would be easier to control and study. If a graphical display for the simulation were to be developed, then the evolution of interesting behaviors could be demonstrated visually. Extensions of these types would also make laboratory style analysis much easier.

By using the tools of Artificial Life as a teaching tool, we not only explain the wonders of life as we know it, but provide examples of general principles of life. We not only give examples of what already exists, but create new possibilities never yet seen before. Artificial Life enables us to think in new ways about life. It abstracts living processes and can establish new precedents for what a living system is. This simulation is one example of the power of creation that Artificial Life embodies. With these tools at hand, we can imagine an unlimited number of possible worlds, and in doing so shift our focus from *life-as-we-know-it* to *life-as-it-could-be.*

Appendix A

Extra Code

A.1 Standard Code Extensions

The functions defined in this module were made specifically for this project, but are so general in nature that they could very well serve as basic extensions to the Haskell language, and be used outside of this project.

```
>module StandardExts
> (partition,remove,
> convert) where
```

The first function is similar to the Haskell function `filter`, which takes a list and removes all elements that do not satisfy a predicate. Instead of removing these elements, the `partition` function places them in another list, thus partitioning the list into two lists: one list of elements that satisfy the predicate and one list of elements that do not satisfy the predicate. The end result is s tuple of two lists, which is built by anonymous functions in the function's two case checks.

Inputs:

- predicate that takes a type `a` as input
- list of type `a`

```
>partition ::  (a → Bool) → [a] → ([a],[a])
>partition f [] = ([],[])
>partition f (a:as)
> | f a = (λ x (xs,ys) → (x:xs,ys)) a (partition f as)
> | otherwise = (λ y (xs,ys) → (xs,y:ys)) a (partition f as)
```

Another very general list function is `remove`, which removes a given element from a list if it is found. Since the element to be removed may or may not be in the list, a Boolean value is returned with the output indicating whether or not the element was found and removed. If the element is not found, then the list returned is the original unaltered list. Note that `remove` only finds the first instance of an element in the list. Should other instances exist, they will remain. This is not a problem in the simulation, because `remove` is used on lists of organisms, which all have unique ID numbers. A list is incapable of having two of the same organism. Another important requirement of the function is that an instance of `Eq` exists for whatever data type the list contains.

Inputs:

- element of type `a` to remove
- list of type `a` to remove from (an instance of `Eq` must exist for `a`)

```
>remove :: Eq a ⇒ a → [a] → ([a],Bool)
>remove x [] = ([],False)
>remove x (y:ys)
> | x ≡ y = (ys,True)
> | otherwise = (λ a (bs,c) → (a:bs,c)) y (remove x ys)
```

The last function of this module is a very simple one that converts numbers from `Integral` types to any other numerical type. For example, it could be used to convert the `Int` value 3 into the `Float` value 3.0. The type that the output is converted to depends on what is needed in the function from which it is called. The output will automatically be of the correct numerical format so as to avoid type conflicts with the function that calls it.

Inputs:

- an `Integral` type value

```
>convert :: (Integral a, Num b) ⇒ a → b
>convert = fromInteger.toInteger
```

A.2 Statistics

Tools from statistics help in analyzing large amounts of data. Statistical functions provide macro level information about a set of data. However, care must be taken in interpreting statistical data. Only when considering information at both macro and micro levels can the most accurate interpretation be produced.

The functions of this module are designed for use on lists of numbers. Numbers are commonly of the `Integral` or `Floating` type. These two number types must be handled differently, so there are two versions of every function presented in this module: one for `Integral` types and one for `Floating` types.

The number of functions presented in this module is decidedly limited, due to the fact that most of the complex analysis of data is performed by Microsoft Excel, which has its own suite of statistical capabilities. Data reports, as explained in *Report*, are designed to have a format that can be easily copied into Excel, where other statistical calculations can be performed, and graphs can be made. It is however useful to have a few statistical calculations handled by Haskell instead of Excel, which is the purpose of this module.

```
>module Statistics
> (averageIntegral, averageFloating,
> varianceIntegral, varianceFloating,
> standardDeviationIntegral, standardDeviationFloating) where

>import StandardExts -- Standard Code Extensions
```

One of the most common and basic statistical measures is the “arithmetic average”, also known as the mean. This is a measure of central tendency. The average of a list of numbers with length `n` is the sum of those numbers divided by `n`. Regardless of whether the list contains integers or floating point values, the average of the list is a floating point number. For this reason both the `Integral` and `Floating` type average functions return `Floating` type output.

The `averageIntegral` and `averageFloating` functions are essentially the same, except that in the `averageIntegral` function, the sum of the list elements is converted to a floating point value before the division operator is used. Both functions also convert the length of the list `n` to a floating point value for the same reason.

Inputs:

- list of `Integral` type numbers

```
>averageIntegral ::  (Integral a, Floating b) ⇒ [a] → b
>averageIntegral xs = ((convert.sum) xs) / ((convert.length) xs)
```

Inputs:

- list of `Floating` type numbers

```
>averageFloating ::  Floating a ⇒ [a] → a
>averageFloating xs = (sum xs) / ((convert.length) xs)
```

Other common measures of central tendency that are not included are the median and mode. These measures are generally less useful. For the rare cases in which they are needed, they can be calculated by Excel.

The next set of functions measure variability, which is how different the data points are from each other. Common measures of variability are "variance" and the closely related measure "standard deviation". Standard deviation is defined differently on sample sets than on entire populations, so care is taken to choose the proper definition below. The sample standard deviation is an unbiased estimator for the standard deviation of the entire population, but since we have full access to data on the entire population, we use the population definition instead.

All of these definitions make use of the sum of the squared deviations of the data points. The "deviation" of a data set is the distance between a point and the mean of the set. The `squareDeviationsSumIntegral` and `squareDeviationsSumFloating` functions calculate the sum of the squares of each deviation. There are some small differences between them: they use different functions to calculate the mean (`averageIntegral` and `averageFloating`), and `squareDeviationsSumIntegral` uses the `convert` function from *Standard Code Extensions* to change the integer data points to floating point values. Each function uses one `map` to subtract the mean from each data point, and another `map` to square all of the resulting differences. Then `sum` is used to get the final result.

Inputs:

- list of `Integral` type numbers

```
>squareDeviationsSumIntegral ::  (Integral a, Floating b) ⇒ [a] → b
>squareDeviationsSumIntegral xs = (sum (map (^2) deviations))
> where
>    mean = averageIntegral xs
>    deviations = map (((λ m x → (x - m)) mean).convert) xs
```

Inputs:

- list of `Floating` type numbers

```
>squareDeviationsSumFloating ::  Floating a ⇒ [a] → a
>squareDeviationsSumFloating xs = (sum (map (^2) deviations))
> where
>    mean = averageFloating xs
>    deviations = map ((λ m x → (x - m)) mean) xs
```

The "variance" is defined as the sum of the squared deviations divided by the number of data points. That makes it the average squared distance from the mean, and as such a measure of how spread out the data points are. Given the results from `squareDeviationsSumIntegral` and `squareDeviationsSumFloating` it is very easy to calculate `varianceIntegral` and `varianceFloating` by dividing these results by the lengths of the lists of data. Technically, this definition is known as the "population variance", which is consistent with our decision made above. We choose not to define the "sample variance", which is obtained by dividing by $(n-1)$ instead of n, where n is the number of data points.

Inputs:

- list of `Integral` type numbers

```
>varianceIntegral ::  (Integral a, Floating b) ⇒ [a] → b
>varianceIntegral xs = (squareDeviationsSumIntegral xs) / ((convert.length) xs)
```

Inputs:

- list of `Floating` type numbers

```
>varianceFloating ::  Floating a ⇒ [a] → a
>varianceFloating xs = (squareDeviationsSumFloating xs) / ((convert.length) xs)
```

With the variance, one can easily calculate the "standard deviation". Standard deviation is yet another measure of variability, but it has the added bonus of being measured in the same units that the data points are measured in, unlike variance, which is measured in units squared. The standard deviation is the square root of the variance. As with variance, this definition conforms to the population definition of standard deviation, rather than the sample set definition.

Inputs:

- list of `Integral` type numbers

```
>standardDeviationIntegral ::  (Integral a, Floating b) ⇒ [a] → b
>standardDeviationIntegral xs = √(varianceIntegral xs)
```

Inputs:

- list of `Floating` type numbers

```
>standardDeviationFloating ::  Floating a ⇒ [a] → a
>standardDeviationFloating xs = √(varianceFloating xs)
```

A.3 Data Analysis

The simulation produces an enormous amount of data. For every time step a text file with information about the state of the world is created. So much data is created, that it cannot be easily analyzed in raw form. The functions in this module extract meaningful data from the raw data and cast it in a form that is understandable.

Before the data in the text output of the simulation can be properly processed, it must be read, parsed and stored in a form that Haskell can understand and manipulate. For this purpose several data types are defined in this module which are similar to data types used in the simulation. These are the `Moment`, `Region` and `Profile` data types, which correspond to the `BinCollection`, `Bin` and `Animal` data types respectively.

The functions defined for interpreting and recasting the data from the simulation are all pure functions (they have no functional side-effects), but because the simulation data comes from text files, some functionally impure `IO` operations are needed to process the data. Several general `IO` functions are defined at the end of this module that take one of the pure interpreting functions as input and apply it to data taken from a file or files. This keeps the amount of impure imperative style code to a minimum.

Many of the data processing functions in this module have two versions: one for a single `Region` (`Bin`) and one for the entire `Moment` (`BinCollection`). By convention, the names of functions that only return information about a single `Region` end in `R`, and the names of functions that return information about all `Region`'s in a given `Moment` end in `A`.

```
>module Analyze
> (module Statistics,
> Moment, time, world,
> Region, population, foliage,
> Profile, idN, position, energy, heading, age, traits,
> readMoment,
> animalsR, animalsA,
> plantsR, plantsA, plantPositionsR, plantPositionsA,
> animalActionsR, animalActionsA,
> ActionData, passedAway, starved, rotted, eaten, born,
>    nothing, mated, atePlant, predated, cannibalized,
>    ateCarrion, failedMating,
> populationProfilesA, populationProfilesR, liveProfilesA, liveProfilesR,
> deadProfilesA, deadProfilesR, livePropertiesA, livePropertiesR,
> deadPropertiesA, deadPropertiesR, generalPropertiesA, generalPropertiesR,
> animalMatingCountdownsA, animalMatingCountdownsR,
> traitsA, traitsR,
> deriveSex, deriveDiet, deriveStartingEnergy, deriveSight, deriveManeuverability,
> deriveMoveSpeed, deriveLifespan, deriveMutationRate, deriveRadius, deriveMatingCost,
> deriveLearningRate, deriveMaturity, deriveMatingTime,
> deriveDummyTrait, deriveMatingRecovery,
> animalSexesA, animalSexesR, animalDietsA, animalDietsR,
> animalMaturitiesA, animalMaturitiesR, animalMatingTimesA, animalMatingTimesR,
> EatingData, none, herb, carn, cann, carr, herbCarn, herbCann, herbCarr,
>    carnCann, carnCarr, cannCarr, herbCarnCann, herbCarnCarr,
>    herbCannCarr, carnCannCarr, herbCarnCannCarr,
> traitA, traitR,
> deriveSpecies, speciesEq,
> groupSpecies,
> Report, total, live, dead, plantCount, action, male, female,
>    eating, mature, immature, matingReady, matingRecovering,
>    energies, ages, matingCountdowns, maximumEnergies, deathPoints,
>    startingEnergies, sights, maneuverabilities, maximumSpeeds,
>    lifespans, mutationRates, radii, matingCosts, learningRates,
>    maturityAges, dummyTraits, matingRecoveries,
> reportCountsA, reportCountsR,
> reportStatisticsA, reportStatisticsR,
> byRegion,
> fromTimeRange,
> command,
> showIO, writeIO,
> module Animal, module Bins, module Plants) where
```

Some functions from the *Animals* section are needed in this module, but still others are replaced in this module by functions of the same name. The functions in this module serve essentially the same purposes as the functions whose names they have stolen, but they are used in conjunction with instances of `Profile` instead of instances of `Animal`. Because the functions do the same work, the names are reused, which requires the hiding of the original functions from the *Animals* section. The other modules listed below are also used in this module.

```
>import Animal hiding (idN, position, heading, age, energy, traits,
>   lastAction, matingCountdown, maxEnergy, deriveSpecies, speciesEq, deathPoint)
>   -- Animals
>import MyBRAMatrix -- Binary Random Access Matrix
>import Environment -- Environment
>import Plants -- Plants
>import Bins -- Bins
>import Statistics -- Statistics
>import StandardExts -- Standard Code Extensions
>import Constants -- Constants
```

The Moment data type is a collection of all the data from one output text file from the simulation. It represents every important aspect of the simulation on a given time step. Because of this it is similar to a BinCollection, which represents the current simulation state during execution. Instead of holding a BRAMatrix of Bin's, a Moment holds a BRAMatrix of Region's, which are this module's equivalent for Bin's. A Moment is also different from a BinCollection in that it records its time step. A BinCollection only holds the current state of execution, and contains no information about how many time steps have passed. A Moment knows when it occurred in relation to other Moment's.

```
>data Moment = M {time::Integer, world::BRAMatrix Region}
>    deriving (Show, Eq)
```

A Region holds the information that a Bin holds, except that the list of Animal's is replaced with a list of Profile's. The PlantGrid that each Bin has remains a PlantGrid in the definition of Region (the PlantGrid's data member is foliage). Unlike a Bin, Region's have two constructors: Null and R. A Null instance holds no data, which seems unnecessary since all Bin's hold information in them. The Null instance does not correspond to any Bin in the simulation, but is instead used only as a temporary place holder when building a Moment instance from file information. When creating a Moment, the world data member, the BRAMatrix of Region's, first has a Null Region in every index. The initialize2D function requires a default value for each index and Null is this value. Before any Moment is processed by a data collecting function, all instances of Null are replaced by instances of R type Region's.

```
>data Region
> = Null
> | R {population::[Profile], foliage::PlantGrid}
>    deriving (Show, Eq)
```

An instance of the Profile data type stores information about an instance of the Animal data type. However, a Profile stores slightly less data than an Animal. A Profile holds all information about a single Animal that is accessible from a data file that was output by the simulation. This means that a Profile does not store information about an Animal's neural network, and rather than have an instance of Traits, a Profile only stores the traits chromosome of the Animal. All other data members are the same, and even have the same names.

Like the Animal data type, the Profile data type has both a Live instance and a Dead instance. Because only living Animal's have neural networks and traits, an instance of a dead Profile is identical to an instance of a dead Animal. Both types use the same constructor and data member names.

```
>data Profile
> = Live {idN::Integer, position::(Float,Float), heading::Float, age::Int,
>    energy::Float, traits::[Float], lastAction::Int, matingCountdown::Int,
>    maxEnergy::Float, deathPoint::Float}
> | Dead {idN::Integer, position::(Float,Float), energy::Float, lastAction::Int}
>    deriving Show
```

Equality between `Profile`'s is defined in the same way as it is between `Animal`'s. Two `Profile`'s are equal if they have the same ID number, meaning that any given `Profile` in any given `Moment` is only equal to itself. Of course, the same ID number can occur in different `Moment`'s if the `Animal` was in existence at both given `Moment`'s.

```
>instance Eq Profile where
>    p1 ≡ p2 = (idN p1) ≡ (idN p2)
```

The above data structures are used to represent data derived from output text files of the simulation. In order to create them, one must read and parse data from these files. A parser works by reading a `String` or `String`'s of data, extracting and recasting into a suitable format some information from this data, and then returning the as yet unprocessed input to be further processed. The `reads` function provided by Haskell reads data of a specified format from the beginning of a `String`. The substring of data in the `String` that is not of the desired format is returned by the function to be further processed. The following functions are designed to do this, starting with `readMoment`. The `readMoment` function is at the highest level of parsing. It parses and returns an instance of `Moment` by calling other functions that build the smaller individual components of a `Moment`.

Because this function reads from a file, it returns an `IO` type. Furthermore, there is a chance that the file to be read does not exist, in which case an exception is thrown. To work around this a `Maybe` type is used to hold the `Moment` instance. If reading from the file is successful, then the `Just` instance of the `Moment` is returned. Otherwise `Nothing` is returned. The file that the function attempts to read is defined in terms of the `Integer` sent as input to the function. This input is the number of the file to be read. Every output file is named "data" followed by the time step on which the file was created. Every output file has the extension "txt". This file is read by the `readFile` command and the result, if no exception occurs, is stored in `input`. This is sent to the `makeRegions` function, which starts the actual parsing process.

Inputs:

- number of the file to read from (the time step on which the file was written)

```
>readMoment ::  Integer → IO (Maybe Moment)
>readMoment n = catch oMom (λ e → return Nothing)
> where
>    oMom ::  IO (Maybe Moment)
>    oMom =
>       do
>          input ← readFile (``data/data''++(show n)++``.txt'')
>          let rs = makeRegions input
>          return (Just (M n rs))
```

The `makeRegions` function returns a `BRAMatrix` of `Region`'s that become the `world` entry for a `Moment` instance. A new `BRAMatrix` with the default `Region` type of `Null` stored in every index is created and sent as input to the helper function `go` along with the text data from the input file. The text input is converted from a `String` to a list of `String`'s by the `lines` function. Every call to the `go` function updates one of the `Null` type regions to an `R` type `Region` containing the data from one of the `Bin`'s of the simulation.

The list of input `String`'s holds one `String` for every line of data in the text file it came from. Once the number of lines left in the list drops to one, all that is left in the list is a return carriage. In this case the fully updated `BRAMatrix` is returned. Otherwise there is still data left to parse, but the number of lines that describe a single `Region` is variable. It depends on the number of `Animal`'s in the `Region`. In order to get the data needed to update a `Null Region` to an `R Region`, the `parseRegion` function is used. The `parseRegion` function returns an `R` type instance of `Region`, its index within the `BRAMatrix`, and whatever is left from the list of input `String`'s. This list is used in the recursive call to `go` so that data for the rest of the `Region`'s can be parsed and applied to the `BRAMatrix`.

Inputs:

- String of all data from a single output file from the simulation

```
>makeRegions ::  String → BRAMatrix Region
>makeRegions s = go g d
> where
>    g = initialize2D binGridSize Null
>    d = lines s

>    go ::  BRAMatrix Region → [String] → BRAMatrix Region
>    go rg ss
>       | length ss > 1 = go (update2D p r rg) remains
>       | otherwise = rg
>          where
>             (p,r,remains) = parseRegion ss
```

The parseRegion function receives a list of String's as input and uses it to create a single R type Region instance. The function does not know in advance how many lines of the input list describe the Region it wants to parse, so it goes through the input line by line searching for special formatting queues in the input.

The first line of input always holds the index in the BinGrid of the Bin being used to form the Region instance. This index becomes the index of the Region within the Moment's world data member. The next several lines of data contain information about Animal's. Specifically, each line holds data on one Animal. The parseProfiles function parses each of these lines into a Profile and returns the list of Profile's, which becomes the population of the Region. The function also returns the rest of the input lines that did not contain Animal data. All that remains in this list is information about the Region's PlantGrid, which gets sent to and parsed by parsePlants. This function returns a PlantGrid, which becomes the Region's foliage, as well as the rest of the input String's. The rest of the data in this list (if anything is left) holds data for the rest of the Region's.

A 3-tuple is returned by the function. It holds the position of the parsed Region in the world BRAMatrix, the parsed Region itself, and any remaining input data.

Inputs:

- list of String's where each String is a line of input from a single output file from the simulation

```
>parseRegion ::  [String] → ((Int,Int), Region, [String])
>parseRegion (p:ps) = (index, R profiles plants, remains)
> where
>    [(index,_)] = (reads p)::[((Int,Int),String)]
>    (profiles,rest) = parseProfiles ps
>    (plants,remains) = parsePlants rest
```

The first part of the data sent to the function parseProfiles contains information for one Profile per line. The separation between the Animal data and plant data is a blank line. This line contains a carriage return however, so the break between the Animal and plant input is detected when the length of the String being processed is at most one. This is the base case that causes the function to return the leftover input data along with the Profile's it has created. On every recursive step of the function a single line of text is parsed and turned into a Profile by parseOneProfile. Because a tuple is returned by the function, an anonymous function is used to add the Profile's to the output as the recursive calls collapse.

Inputs:

- list of String's where each String is a line of input from a single output file from the simulation (starting with Animal data)

```
>parseProfiles :: [String] → ([Profile],[String])
>parseProfiles (s:ss)
> | length s > 1 = (λ p (ps,s) → (p:ps,s)) (parseOneProfile s) (parseProfiles ss)
> | otherwise = ([],ss)
```

Information about Animal's is printed to output files by the niceShowAnimal function. This function formats the data in a way that is relatively easy to parse. The first character tells whether the Animal is living (the letter L) or dead (the letter D). These two cases are handled differently by parseOneProfile. A living Animal has all of the data fields a dead Animal has and more. This common information is listed first for both Animal types, so that the first four values can be parsed in the same manner. In order, these values are ID number, position, energy and last action. Following these values, a living Animal also has its age, heading, mating countdown, maximum energy, death point and traits chromosome.

All data is converted from a String to its appropriate type by a reads command followed by a format specification. Strangely, reads returns a tuple within a single element list. The first value in the tuple is the result of converting the characters at the beginning of the String into the desired format. The second value in the tuple is the rest of the String that was not converted. Because of the way the data is formatted, the first character in this String will be a blank space as long as there is data left to parse, so the first character is immediately dropped from what is returned. This result is then parsed again to get the next value from the String. When the last action value is returned, the first character of the leftover String is not dropped because if the Animal is of the Dead type there is no character to drop. If however the Animal is alive, then the parsing continues on the tail of the String leftover after the last action was retrieved. Once the last value (traits chromosome) in the String is reached, there is no data left in the String to parse. The values generated by the series of reads calls are used to make a Profile of the appropriate type (Live or Dead), which is returned.

Inputs:

- a String that is a line of input from a single output file from the simulation (the String describes an Animal)

```
>parseOneProfile :: String → Profile
>parseOneProfile (s:_:ss)
> | s ≡ 'L' =
>   Live {idN = id, position = pos, energy = en, age = ag,
>      heading = h, traits = ts, lastAction = la,
>      matingCountdown = mc, maxEnergy = me, deathPoint = dp}
> | s ≡ 'D' = Dead {idN = id, position = pos, energy = en, lastAction = la}
> | otherwise = error ''Not an animal''
>   where
>      [(id,(_:r1))] = (reads ss)::[(Integer,String)]
>      [(pos,(_:r2))] = (reads r1)::[((Float,Float),String)]
>      [(en,(_:r3))] = (reads r2)::[(Float,String)]
>      [(la,r4)] = (reads r3)::[(Int,String)]
>      [(ag,(_:r5))] = (reads (tail r4))::[(Int,String)]
>      [(h,(_:r6))] = (reads r5)::[(Float,String)]
>      [(mc,(_:r7))] = (reads r6)::[(Int,String)]
>      [(me,(_:r8))] = (reads r7)::[(Float,String)]
>      [(dp,(_:r9))] = (reads r8)::[(Float,String)]
>      [(ts,_)] = (reads r9)::[([Float],String)]
```

The plant data found in the output files is produced by the function `niceShowBRAMatrix`. `PlantGrid`'s are `BRAMatrix`'s storing Boolean values, so when `niceShowBRAMatrix` is used on a `PlantGrid`, several lists of Boolean values are the output. Each line of output holds a list of Boolean values, which are the contents for a given row in the `PlantGrid`.

The `parsePlants` function takes these lines of output and makes a `PlantGrid` from them. It starts by using `initialize2D` to create a new `PlantGrid` with `False` for every index. The `go` helper function steps through each line of input data, and updates the corresponding row. The `go` function parses each line into a list of Boolean values and sends this as input to its own helper function `uRow`, which processes the contents of individual rows. While stepping through the list of Boolean values the function ignores all instances of `False` and only updates the `PlantGrid` when `True` is found. The `uRow` function tracks the current column by maintaining its own counter, and uses the `go` function's row counter, with its own, to specify the position to update.

Besides parsing and returning a `PlantGrid` from the input, the function also returns the leftover input `String`'s to be further parsed.

Inputs:

- list of `String`'s that describe a `PlantGrid`

```
>parsePlants :: [String] → (PlantGrid,[String])
>parsePlants ps = go 0 ps (initialize2D (arrayWidth,arrayHeight) False)
> where
>   go :: Int → [String] → PlantGrid → (PlantGrid,[String])
>   go _ [] g = (g,[])
>   go y (s:ss) g
>      | length s > 1 = go (y+1) ss (uRow 0 row g)
>      | otherwise = (g,tail ss)
>        where
>           [(row, _ )] = (reads s)::[([Bool],String)]

>           uRow :: Int → [Bool] → PlantGrid → PlantGrid
>           uRow _ [] g = g
>           uRow x (b:bs) g
>              | b = uRow (x+1) bs (update2D (x,y) b g)
>              | otherwise = uRow (x+1) bs g
```

Once a `Moment` has been parsed it can be analyzed by one of the many functions below. As mentioned above, for each function there is a version that analyzes all `Region`'s and one that analyzes a single `Region`. All `Region` type functions take a `Moment` and the index of a `Region` as input, and end with an `R`. Functions that analyze the entire world take a `Moment` as input and end with the letter `A`.

Any function that returns data on the `Animal` population must first collect their `Profile`'s. The two functions below collect population `Profile`'s. They are useful by themselves, but are mostly used by other functions to retrieve `Profile`'s, from which more specific information is derived. The `populationProfilesA` function uses `elems2D` to get a list of all `Region`'s, maps `population` across this list to get a list of lists of `Profile`'s (each list is the population from one `Region`) and then folds the append operation across this list to combine all `Profile`'s into a single list, which is returned. The `populationProfilesR` function returns the `population` data member of the `Region` returned by `lookup2D`.

Inputs:

- a `Moment`

```
>populationProfilesA :: Moment → [Profile]
>populationProfilesA m = foldl1 (++) (map population (elems2D (world m)))
```

Inputs:

- index of the Region to take data from
- a Moment

```
>populationProfilesR :: (Int,Int) → Moment → [Profile]
>populationProfilesR p m = population (lookup2D p (world m))
```

A list of Profile's can be further divided into Live and Dead types. The information that can be gathered from an Animal depends on whether it is living or dead, so it is important to sort the Profile's into these two distinct types. At the level of a single Profile, the two predicates that test for living and dead Animal's are isLive and isDead respectively. The isLive function returns True if a Profile is of the Live type and False otherwise. The isDead function does the opposite.

Inputs:

- Profile

```
>isLive :: Profile → Bool
>isLive (Live{}) = True
>isLive _ = False
```

Inputs:

- Profile

```
>isDead :: Profile → Bool
>isDead (Dead {}) = True
>isDead _ = False
```

When used with the filter function on a list of Profile's, these predicates reduce that list so that it contains only one type (Live or Dead). The functions below apply the filter function in this fashion. They use a list of Profile's retrieved by applying either populationProfilesA or populationProfilesR. the liveProfilesA function returns Live type Profile's from the whole world, liveProfilesR returns Live Profile's from one Region, deadProfilesA returns Dead Profile's from the whole world, and deadProfilesR returns Dead Profile's from a single Region.

Inputs:

- Moment

```
>liveProfilesA :: Moment → [Profile]
>liveProfilesA m = filter isLive (populationProfilesA m)
```

Inputs:

- index of the Region to take data from
- Moment

```
>liveProfilesR :: (Int,Int) → Moment → [Profile]
>liveProfilesR p m = filter isLive (populationProfilesR p m)
```

Inputs:

- `Moment`

```
>deadProfilesA :: Moment → [Profile]
>deadProfilesA m = filter isDead (populationProfilesA m)
```

Inputs:

- index of the `Region` to take data from
- `Moment`

```
>deadProfilesR :: (Int,Int) → Moment → [Profile]
>deadProfilesR p m = filter isDead (populationProfilesR p m)
```

There are also situations when both types of `Profile`'s are needed, but they must be separated. The `partitionProfiles` function takes care of this with the help of the `partition` function. A tuple with a list of `Live` type `Profile`'s and a list of `Dead` type `Profile`'s is returned. Note that the `isLive` predicate is used. Either the `isLive` or `isDead` predicate would work, because these are the only types of `Animal`'s. Once all of one type is removed from a list, everything left in the list must be of the other type.

Inputs:

- list of `Profile`'s

```
>partitionProfiles :: [Profile] → ([Profile],[Profile])
>partitionProfiles = partition isLive
```

One of the most basic operations that can be performed using the above functions is to count `Animal`'s of the different types. The number of total `Animal`'s in the population at any time is the sum of the living and dead `Animal`'s. The `animalsR` and `animalsA` functions count `Animal`'s in `Region`'s and the whole world respectively. They both return a 3-tuple as output. The three numbers in the tuple are the total number of `Animal`'s, the number of living `Animal`'s and the number of dead `Animal`'s. Both functions make use of `partitionProfiles`. The only difference between them is the function they use to collect `Animal`'s `Profile`'s (for a `Region` or for the whole world).

Inputs:

- index of the `Region` to take data from
- `Moment`

```
>animalsR :: (Int,Int) → Moment → (Int,Int,Int)
>animalsR p m = (live+dead, live, dead)
>   where
>      (l,d) = partitionProfiles (populationProfilesR p m)
>      live = length l
>      dead = length d
```

Inputs:

- Moment

```
>animalsA :: Moment → (Int,Int,Int)
>animalsA m = (live+dead, live, dead)
>   where
>      (l,d) = partitionProfiles (populationProfilesA m)
>      live = length l
>      dead = length d
```

Counting the plant population is also important. A plant is a value of `True` stored in a `PlantGrid`. To count them, the contents of a `PlantGrid` are converted to a list and then filtered with the `id` function, leaving only `True` values. The length of the resulting list equals the number of plants in the list. Where this list comes from is different when counting plants in one `Region` or from all `Region`'s. Both versions of the function use `elems2D` to put the contents of `PlantGrid`'s into lists. The version that works on the whole world folds an append operation across the list of lists from each `Region` (created with `map`) to make one list for the entire world.

Inputs:

- index of the `Region` to take data from
- Moment

```
>plantsR :: (Int,Int) → Moment → Int
>plantsR p (M {world = g}) = length (filter id (elems2D (foliage (lookup2D p g))))
```

Inputs:

- Moment

```
>plantsA :: Moment → Int
>plantsA (M {world = g}) =
>   length (filter id (foldl1 (++) (map elems2D (map foliage (elems2D g)))))
```

The only property that plants have is position, so in analyzing the plant population it is important to have access to this data. The following functions return lists of positions of plants. The `plantPositionsR` function counts plants in a single `Region`. First it looks up the specified `Region` with `lookup2D` and then retrieves its `PlantGrid` from the `foliage` data member. The `assocs2D` function extracts a list of position-value pairs from the grid. The `snd` function filters out all tuples with a value of `False`. This works because the second value in every tuple is a Boolean value. What remains is a list of tuples where every first entry is the position of a plant and every second position is `True`. The `fst` function is mapped across this list to drop the Boolean part, leaving only a list of plant positions.

The function `plantPositionsA` returns the positions of plants in the entire world, and it works by making a call to `plantPositionsR` for each `Region`, and combining the results. Because plant positions are given relative to the `Region` they are in, steps are taken to prevent the positions from different `Region`'s from overlapping with each other. All plants in `Region` (0,0) have the same position in their `Region` as they do in the world as a whole. Plants from the other `Region`'s have their world positions offset by positive multiples of `arrayWidth` and `arrayHeight`. More specifically, an x-coordinate x_w in the world is x_r + (`arrayWidth` $\times$ x_g), where x_r is the x-coodinate within the `Region` and x_g is that `Region`'s x-coordinate within the `BRAMatrix` of `Region`'s. The world y-coordinate is calculated similarly with `arrayHeight` instead of `arrayWidth`.

A helper function named `go` is used to recursively traverse every index in the grid of `Region`'s. This traversal is in row major order, so the x-coordinate is incremented in the recursive call of the case where data is gathered. This is the `otherwise` case. When the x-coordinate goes out of bounds, meaning it equals `binGridSizeX`, it is reset to 0 on the recursive call and the y-coordinate is incremented. The base case comes when the y-coordinate goes out of bounds to equal `binGridSizeY`. Data is gathered in the `otherwise` case with a call to `plantPositionsR`. Every position in the list returned by this function has its coordinates offset as described above by an anonymous function, which is mapped across the list. The resulting list is appended to the recursive call to `go`.

Inputs:

- index of the `Region` to take data from
- `Moment`

```
>plantPositionsR ::  (Int,Int) → Moment → [(Int,Int)]
>plantPositionsR p (M {world = g})
>    = map fst (filter snd (assocs2D (foliage (lookup2D p g))))
```

Inputs:

- `Moment`

```
>plantPositionsA ::  Moment → [(Int,Int)]
>plantPositionsA m = go (0,0)
> where
>   go ::  (Int,Int) → [(Int,Int)]
>   go (x,y)
>      | x ≡ binGridSizeX = go (0,y+1)
>      | y ≡ binGridSizeY = []
>      | otherwise =
>        (map (λ (x1,y1) → (x1 + (x × arrayWidth), y1+(y × arrayHeight)))
>           (plantPositionsR (x,y) m))++(go (x+1,y))
```

Unlike plants, `Animal`'s have many properties. These properties are stored in `Profile`'s and accessed through a `Profile`'s data members. Even in this form the data must be further processed so that analysis of it is tractable. By isolating individual properties from `Profile`'s, one can better interpret the data. The following functions take as input a function that produces information about a `Profile`. Data member functions all fall into this category, though any function that takes a `Profile` as its only input can be used. This input function is then applied across a list of `Profile`'s in a `Moment`, either for the entire world or in a single `Region`, and the list of values derived is the output.

The functions also distinguish between the type of `Profile` they gather data from. There are functions that only gather data from `Live` type `Profile`'s, ones that only gather data from `Dead` type `Profile`'s and ones that gather data from both types. All of these functions work by mapping the function they receive as input across a list of `Profile`'s returned by another function, which returns `Profile`'s of the type that is of interest. For example, `livePropertiesA` returns information about `Live` type `Profile`'s for the entire world, so it uses the `liveProfilesA` function to gather `Profile`'s. The other functions are similar. Each gathers data for either the entire world or a single `Region`, and uses one of the following types of `Profile`'s: `Live`, `Dead` or both.

Inputs:

- function that generates data from a `Profile`
- `Moment`

```
>livePropertiesA ::  (Profile → a) → Moment → [a]
>livePropertiesA f m = map f (liveProfilesA m)
```

Inputs:

- function that generates data from a `Profile`
- the index of the `Region` to take data from
- `Moment`

```
>livePropertiesR :: (Profile → a) → (Int,Int) → Moment → [a]
>livePropertiesR f p m = map f (liveProfilesR p m)
```

Inputs:

- function that generates data from a `Profile`
- `Moment`

```
>deadPropertiesA :: (Profile → a) → Moment → [a]
>deadPropertiesA f m = map f (deadProfilesA m)
```

Inputs:

- function that generates data from a `Profile`
- the index of the `Region` to take data from
- `Moment`

```
>deadPropertiesR :: (Profile → a) → (Int,Int) → Moment → [a]
>deadPropertiesR f p m = map f (deadProfilesR p m)
```

Inputs:

- function that generates data from a `Profile`
- `Moment`

```
>generalPropertiesA :: (Profile → a) → Moment → [a]
>generalPropertiesA f m = map f (populationProfilesA m)
```

Inputs:

- function that generates data from a `Profile`
- the index of the `Region` to take data from
- `Moment`

```
>generalPropertiesR :: (Profile → a) → (Int,Int) → Moment → [a]
>generalPropertiesR f p m = map f (populationProfilesR p m)
```

The next set of functions deals with the last action data member of `Animal`'s. The list of last action codes and their meanings from the *Animals* module is repeated here.

Dead `Animal`'s:

-5 : Passed Away A dead animal with this code has died of old age, meaning that its age was equal to its lifespan.

-4 : Starved This animal has run out of energy. More accurately, its energy level has dropped to or below its death point.

-3 : Rotted This animal has already been dead for at least one time step, and now does nothing but rot on every time step.

-2 : Eaten A dead animal with this code has died of being eaten by another animal.

Live `Animal`'s:

-1 : Born This animal has been born, and has yet to perform any actions.

0 : Nothing Besides turning and moving as is required of all living animals, this animal did nothing during the last time step.

1 : Mated This animal mated with another animal to produce offspring during the last time step.

2 : Ate Plant The animal is a herbivore, and ate a plant on the last time step.

3 : Ate Animal of Different Species The animal is a carnivore, and ate a member of another species on the last time step.

4 : Cannibalized Another Animal This cannibal ate a member of its own species on the last time step.

5 : Ate Carrion This carrion eater ate of a corpse on the last time step.

6 : Failed At Mating This animal attempted to mate on the last time step, but for some reason no offspring was produced.

These codes are used to construct a simple data structure called `ActionData`, which stores information about the number of `Animal`'s that performed each of the possible actions listed above on a given time step. This data could be stored in a tuple, but given the number of entries, such a representation would be exceedingly confusing. The `ActionData` data type has one constructor named `A`, and has an entry for each of the action codes listed above. The only data member whose name is perhaps not obvious is `predated`, which counts the number of `Animal`'s that ate `Animal`'s of other species (code 3).

```
>data ActionData
> = A {passedAway::Int, starved::Int, rotted::Int, eaten::Int,
>   born::Int, nothing::Int, mated::Int, atePlant::Int, predated::Int,
>   cannibalized::Int, ateCarrion::Int, failedMating::Int}
>      deriving Show
```

Instances of `ActionData` are created by `animalActionsA` and `animalActionsR`, both of which use `animalActionsCount` to do the counting. Both `animalActionsA` and `animalActionsR` generate a list of action codes and send it to `animalActionsCount` as input. The `animalActionsCount` function uses a helper function `accum` and a large tuple accumulator to store information from the list as it is processed. The `accum` function pattern matches against every possible action code and increments the appropriate value in the tuple accumulator. Once the list of action codes is empty, the accumulator is returned. The returned accumulator values are then put in their proper places in the `ActionData` constructor.

```
>animalActionsCount ::  [Int] → ActionData
>animalActionsCount as = A a b c d e f g h i j k l
> where
>   (a,b,c,d,e,f,g,h,i,j,k,l) = accum as (0,0,0,0,0,0,0,0,0,0,0,0)

>   accum ::  [Int] → (Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int)
>         → (Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int)
>   accum [] ad = ad
>   accum (x:xs) (a,b,c,d,e,f,g,h,i,j,k,l)
>      | x ≡ (-5) = accum xs (a+1,b,c,d,e,f,g,h,i,j,k,l)
>      | x ≡ (-4) = accum xs (a,b+1,c,d,e,f,g,h,i,j,k,l)
>      | x ≡ (-3) = accum xs (a,b,c+1,d,e,f,g,h,i,j,k,l)
>      | x ≡ (-2) = accum xs (a,b,c,d+1,e,f,g,h,i,j,k,l)
>      | x ≡ (-1) = accum xs (a,b,c,d,e+1,f,g,h,i,j,k,l)
>      | x ≡ 0 = accum xs (a,b,c,d,e,f+1,g,h,i,j,k,l)
>      | x ≡ 1 = accum xs (a,b,c,d,e,f,g+1,h,i,j,k,l)
>      | x ≡ 2 = accum xs (a,b,c,d,e,f,g,h+1,i,j,k,l)
>      | x ≡ 3 = accum xs (a,b,c,d,e,f,g,h,i+1,j,k,l)
>      | x ≡ 4 = accum xs (a,b,c,d,e,f,g,h,i,j+1,k,l)
>      | x ≡ 5 = accum xs (a,b,c,d,e,f,g,h,i,j,k+1,l)
>      | x ≡ 6 = accum xs (a,b,c,d,e,f,g,h,i,j,k,l+1)
>      | otherwise = error ''Illegal Action''
```

The animalActionsA function generates the list of action codes for animalActionsCount by using generalPropertiesA with lastAction as the Profile information function. The animalActionsR does the same with generalPropertiesR.

Inputs:

- Moment

```
>animalActionsA ::  Moment → ActionData
>animalActionsA m = animalActionsCount (generalPropertiesA lastAction m)
```

Inputs:

- index of the Region to take data from
- Moment

```
>animalActionsR ::  (Int,Int) → Moment → ActionData
>animalActionsR p m = animalActionsCount (generalPropertiesR lastAction p m)
```

Another way to categorize Animal's is in terms of their mating countdowns. One of the factors influencing whether or not an Animal can mate is its mating countdown. It can mate when the countdown is at zero, given that all other requirements are met, and cannot mate otherwise. The animalMatingCountdownsA and animalMatingCountdownsR functions count the number of Animal's in each of these two categories. The counting for both functions is done by animalMatingCountdowns.

The animalMatingCountdowns function uses partition to split the list of mating countdowns into those that are equal to zero and those that are not. The lengths of the resulting lists are returned in a tuple with the count of Animal's with a mating countdown of zero coming first.

Inputs:

- list of mating countdowns taken from the Profile's of living Animal's

```
>animalMatingCountdowns :: [Int] → (Int,Int)
>animalMatingCountdowns ps = (length zero, length notZero)
>   where
>      (zero,notZero) = partition (≡ 0) ps
```

Inputs:

- Moment

```
>animalMatingCountdownsA :: Moment → (Int,Int)
>animalMatingCountdownsA m = animalMatingCountdowns (livePropertiesA matingCountdown m)
```

Inputs:

- index of the Region to take data from
- Moment

```
>animalMatingCountdownsR :: (Int,Int) → Moment → (Int,Int)
>animalMatingCountdownsR p m = animalMatingCountdowns (livePropertiesR matingCountdown p m)
```

While some Animal properties are readily accessible from its Profile, other properties must be derived from the Profile's traits. Profile's do not have a Traits instance. Instead, they only store the traits chromosome, from which all content of a Traits instance can be derived. The next two functions provide access to the traits chromosome by using livePropertiesA and livePropertiesR respectively.

Inputs:

- Moment

```
>traitsA :: Moment → [[Float]]
>traitsA m = livePropertiesA traits m
```

Inputs:

- index of the Region to take data from
- Moment

```
>traitsR :: (Int,Int) → Moment → [[Float]]
>traitsR p m = livePropertiesR traits p m
```

Deriving individual traits from a traits chromosome involves mapping a trait to its proper range and format. The ranges and formats are set by values in the *Constants* module, but the functions that do the derivation are in this module (see below). Any of these functions can be sent as input to traitA or traitR, which derive the values of a single trait for all Animal's in the world or a single Region respectively.

Inputs:

- function that derives a trait value from a traits chromosome
- Moment

```
>traitA :: ([Float] → a) → Moment → [a]
>traitA h m = map h (traitsA m)
```

Inputs:

- function that derives a trait value from a traits chromosome
- the index of the `Region` to take data from
- `Moment`

```
>traitR :: ([Float] → a) → (Int,Int) → Moment → [a]
>traitR h p m = map h (traitsR p m)
```

Every position in the traits chromosome holds information about one trait. When deriving a single trait from a traits chromosome, this value is taken from the rest, which are ignored. The only exception is diet, which is composed of four traits. The trait derivation functions take advantage of the fact that the traits are sorted in a specific order within the chromosome, which allows the proper trait to be found using pattern matching. All unimportant trait genes that precede the gene of interest in the chromosome are ignored. The gene of interest is then converted to its proper range and format using `take01toBool`, `take01toFloat` or `take01toInt`, depending on the trait.

The trait deriving functions are defined below in order of trait gene position in the traits chromosome. The minimum and maximum values for each trait are defined in the *Constants* module and the format of the gene is determined by the function used to map to the trait's range. Boolean traits all have the same range. As mentioned above, the diet trait is derived from four trait genes: one Boolean trait for each eating behavior.

Inputs:

- traits chromosome

```
>deriveStartingEnergy :: [Float] → Float
>deriveStartingEnergy (se:_) = take01toFloat se minStartEn maxStartEn
```

Inputs:

- traits chromosome

```
>deriveSex :: [Float] → Bool
>deriveSex (_:sx:_) = take01toBool sx
```

Inputs:

- traits chromosome

```
>deriveSight :: [Float] → Float
>deriveSight (_:_:s:_) = take01toFloat s minSight maxSight
```

Inputs:

- traits chromosome

```
>deriveDiet :: [Float] → (Bool,Bool,Bool,Bool)
>deriveDiet (_:_:_:c:d:e:f:_) = formDiet (c,d,e,f)
```

Inputs:

- traits chromosome

```
>deriveManeuverability :: [Float] → Float
>deriveManeuverability (_:_:_:_:_:_:m:_)
>   = take01toFloat m minManeuverability maxManeuverability
```

Inputs:

- traits chromosome

```
>deriveMoveSpeed :: [Float] → Float
>deriveMoveSpeed (_:_:_:_:_:_:_:ms:_) = take01toFloat ms minMoveSpeed maxMoveSpeed
```

Inputs:

- traits chromosome

```
>deriveLifespan :: [Float] → Int
>deriveLifespan (_:_:_:_:_:_:_:_:ls:_) = take01toInt ls minLifespan maxLifespan
```

Inputs:

- traits chromosome

```
>deriveMutationRate :: [Float] → Float
>deriveMutationRate (_:_:_:_:_:_:_:_:_:mut:_)
>   = take01toFloat mut minMutationRate maxMutationRate
```

Inputs:

- traits chromosome

```
>deriveRadius :: [Float] → Float
>deriveRadius (_:_:_:_:_:_:_:_:_:_:r:_) = take01toFloat r minRadius maxRadius
```

Inputs:

- traits chromosome

```
>deriveMatingCost :: [Float] → Float
>deriveMatingCost (_:_:_:_:_:_:_:_:_:_:_:mc:_)
>   = take01toFloat mc minMatingCost maxMatingCost
```

Inputs:

- traits chromosome

```
>deriveLearningRate :: [Float] → Float
>deriveLearningRate (_:_:_:_:_:_:_:_:_:_:_:_:lr:_) = lr
```

Inputs:

- traits chromosome

```
>deriveMaturity :: [Float] → Int
>deriveMaturity (_:_:_:_:_:_:_:_:_:_:_:_:_:_:m:_) = take01toInt m minMaturity maxMaturity
```

Inputs:

- traits chromosome

```
>deriveMatingTime :: [Float] → Int
>deriveMatingTime (_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:m:_)
>   = take01toInt m minMatingTime maxMatingTime
```

Inputs:

- traits chromosome

```
>deriveDummyTrait :: [Float] → Float
>deriveDummyTrait (_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:d:_) = d
```

Inputs:

- traits chromosome

```
>deriveMatingRecovery :: [Float] → Int
>deriveMatingRecovery (_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:m:[])
>   = take01toInt m minMatingRecovery maxMatingRecovery
```

Having derived all of these traits, there are now many more ways in which to count and classify `Animal`'s. One of these is by sex. It is important that there be enough members of both sexes in the population to allow for mating to occur. An approximately equal share of both sexes is expected because, barring mutation, one member of each sex is born every time mating occurs, since each offspring gets it sex trait gene from a different parent, and the two parents are of opposite sexes. The `animalSexesA` and `animalSexesR` functions count the number of `Animal`'s of each sex within the entire world and within a single `Region` respectively. Both use the `animalsBySex` function to partition a list of Boolean values, sex traits, into lists of `True` and `False` values, the lengths of which are calculated and returned.

Inputs:

- list of Boolean values which are sex trait indicators

```
>animalsBySex :: [Bool] → (Int,Int)
>animalsBySex bs = (length m, length f)
>   where
>      (m,f) = partition id bs
```

Inputs:

- `Moment`

```
>animalSexesA :: Moment → (Int,Int)
>animalSexesA m = animalsBySex (traitA deriveSex m)
```

Inputs:

- index of the `Region` to take data from
- `Moment`

```
>animalSexesR ::  (Int,Int) → Moment → (Int,Int)
>animalSexesR p m = animalsBySex (traitR deriveSex p m)
```

Another important and more complicated way of sorting `Animal`'s is by their diet. An `Animal`'s diet is defined by four independent Boolean values, which allows for 16 different possible diets. This includes the possibility of an `Animal` evolving a complete lack of eating behaviors, though such an `Animal` would likely not live very long. Because 16 numbers would be confusing in a tuple, the `EatingData` data type is defined below. Like `ActionData` above, and `EatingData` instance is several named data members of the `Int` type with a single constructor. Each possible diet, except for `none`, is named with abbreviations for the eating behaviors that compose it. The abbreviations are "herb" for herbivore, "carn" for eating `Animal`'s of other species, "cann" for cannibal, and "carr" for carrion eater. An `EatingData` instance stores the number of `Animal`'s in a population with each eating type in the following order:

- No eating behavior
- Pure herbivores
- Pure carnivores
- Pure cannibals
- Pure carrion eaters
- Herbivore and carnivore
- Herbivore and cannibal
- Herbivore and carrion eater
- Carnivore and cannibal
- Carnivore and carrion eater
- Cannibal and carrion eater
- Herbivore, carnivore and cannibal
- Herbivore, carnivore and carrion eater
- Herbivore, cannibal and carrion eater
- Carnivore, cannibal and carrion eater
- Herbivore, carnivore, cannibal and carrion eater

```
>data EatingData
> = E {none::Int, herb::Int, carn::Int, cann::Int, carr::Int, herbCarn::Int,
>    herbCann::Int, herbCarr::Int, carnCann::Int, carnCarr::Int, cannCarr::Int,
>    herbCarnCann::Int, herbCarnCarr::Int, herbCannCarr::Int,
>    carnCannCarr::Int, herbCarnCannCarr::Int}
>       deriving Show
```

With `EatingData` defined, it is a simple matter to define `animalDietsA` and `animalDietsR`, which examine populations of `Animal`'s and return the corresponding `EatingData` instances. As with so many other functions in this module, a common counting function is first defined. The `animalsByDiet` function takes a list of diets and uses the helper function `accum` to create `EatingData` from it. The `accum` function has a case for catching each of the 16 possible diet types. The accumulator for `accum` is a tuple, but once returned the data is recast into an `EatingData` instance.

Inputs:

- list of Boolean 4-tuples, each of which represents an `Animal`'s diet

```
>animalsByDiet ::  [(Bool,Bool,Bool,Bool)] → EatingData
>animalsByDiet bs = E a b c d e f g h i j k l m n o p
> where
>   (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p) = accum bs (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

>   accum ::  [(Bool,Bool,Bool,Bool)]
>         → (Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int)
>         → (Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int)
>   accum [] (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p) = (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)
>   accum (x:xs) (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)
>     | x ≡ (False,False,False,False) = accum xs (a+1,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)
>     | x ≡ (True,False,False,False) = accum xs (a,b+1,c,d,e,f,g,h,i,j,k,l,m,n,o,p)
>     | x ≡ (False,True,False,False) = accum xs (a,b,c+1,d,e,f,g,h,i,j,k,l,m,n,o,p)
>     | x ≡ (False,False,True,False) = accum xs (a,b,c,d+1,e,f,g,h,i,j,k,l,m,n,o,p)
>     | x ≡ (False,False,False,True) = accum xs (a,b,c,d,e+1,f,g,h,i,j,k,l,m,n,o,p)
>     | x ≡ (True,True,False,False) = accum xs (a,b,c,d,e,f+1,g,h,i,j,k,l,m,n,o,p)
>     | x ≡ (True,False,True,False) = accum xs (a,b,c,d,e,f,g+1,h,i,j,k,l,m,n,o,p)
>     | x ≡ (True,False,False,True) = accum xs (a,b,c,d,e,f,g,h+1,i,j,k,l,m,n,o,p)
>     | x ≡ (False,True,True,False) = accum xs (a,b,c,d,e,f,g,h,i+1,j,k,l,m,n,o,p)
>     | x ≡ (False,True,False,True) = accum xs (a,b,c,d,e,f,g,h,i,j+1,k,l,m,n,o,p)
>     | x ≡ (False,False,True,True) = accum xs (a,b,c,d,e,f,g,h,i,j,k+1,l,m,n,o,p)
>     | x ≡ (True,True,True,False) = accum xs (a,b,c,d,e,f,g,h,i,j,k,l+1,m,n,o,p)
>     | x ≡ (True,True,False,True) = accum xs (a,b,c,d,e,f,g,h,i,j,k,l,m+1,n,o,p)
>     | x ≡ (True,False,True,True) = accum xs (a,b,c,d,e,f,g,h,i,j,k,l,m,n+1,o,p)
>     | x ≡ (False,True,True,True) = accum xs (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o+1,p)
>     | x ≡ (True,True,True,True) = accum xs (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p+1)
>     | otherwise = error ''Impossible diet''
```

Inputs:

- `Moment`

```
>animalDietsA ::  Moment → EatingData
>animalDietsA m = animalsByDiet (traitA deriveDiet m)
```

Inputs:

- index of the `Region` to take data from
- `Moment`

```
>animalDietsR ::  (Int,Int) → Moment → EatingData
>animalDietsR p m = animalsByDiet (traitR deriveDiet p m)
```

The next trait that can be used to organize `Animal`'s into separate categories is maturity age. The `animalMaturitiesA` and `animalMaturitiesR` functions count `Animal`'s that have reached their maturity ages and those that have not. The common counting function for `animalMaturitiesA` and `animalMaturitiesR` is `animalsByMaturity`, which uses a helper function named `go` to count mature and immature `Animal`'s. It takes a list of `Profile`'s as input and compares each `Profile`'s `age` data member with the maturity age derived from the `Profile`'s traits. The final output is a tuple holding first the number of mature `Animal`'s and second the number of immature `Animal`'s.

Inputs:

- list of `Profile`'s

```
>animalsByMaturity :: [Profile] → (Int,Int)
>animalsByMaturity ps = go ps (0,0)
>   where
>      go :: [Profile] → (Int,Int) → (Int,Int)
>      go [] (a,b) = (a,b)
>      go (p:ps) (a,b)
>         | age p > deriveMaturity (traits p) = go ps (a+1,b)
>         | otherwise = go ps (a,b+1)
```

Inputs:

- `Moment`

```
>animalMaturitiesA :: Moment → (Int,Int)
>animalMaturitiesA m = animalsByMaturity (liveProfilesA m)
```

Inputs:

- index of the `Region` to take data from
- `Moment`

```
>animalMaturitiesR :: (Int,Int) → Moment → (Int,Int)
>animalMaturitiesR p m = animalsByMaturity (liveProfilesR p m)
```

The next trait to sort `Animal`'s by is mating time. `Animal`'s can be sorted by mating time because this trait has a discrete range of values. The size of this range depends on the constants `minMatingTime` and `maxMatingTime`. For this reason, the results of the count are returned in a list instead of a tuple, because a list has variable length. The number of elements in the list equals `maxMatingTime - minMatingTime` + 1, and the counts within the list are ordered from the number of `Animal`'s with `minMatingTime` as their mating time to the number of those with `maxMatingTime` as their mating time.

The common counting function `animalsMatingTimes` creates this list. Its helper function `accum` counts the number of `Animal`'s with a particular mating time on each recursive call. It starts with `minMatingTime` and proceeds from there. All values equal to the current mating time being counted are removed from the overall list using a `filter`, and the resulting list is used in the recursive call. The length of this list subtracted from the length of the original list is the number of occurrences of the given mating time. On the recursive call, the mating time to check is incremented and the length of the remaining list of mating times is recalculated. When the list is empty, then all mating times have been counted. This means that the number of `Animal`'s with a mating time any greater than the last one checked is zero, so the last mating time checked is subtracted from `maxMatingTime`, and this number of zeros is added to the end of the output list.

Inputs:

- list of mating times

```
>animalsMatingTimes ::  [Int] → [Int]
>animalsMatingTimes ts = accum minMatingTime ts (length ts)
>   where
>      accum ::  Int → [Int] → Int → [Int]
>      accum c [] 0 = take (maxMatingTime - c) (repeat 0)
>      accum c ps s = (s - lenN):(accum (c+1) next lenN)
>         where
>            next = filter (≠ c) ps
>            lenN = length next
```

Inputs:

- `Moment`

```
>animalMatingTimesA ::  Moment → [Int]
>animalMatingTimesA m = animalsMatingTimes (traitA deriveMatingTime m)
```

Inputs:

- index of the `Region` to take data from
- `Moment`

```
>animalMatingTimesR ::  (Int,Int) → Moment → [Int]
>animalMatingTimesR p m = animalsMatingTimes (traitR deriveMatingTime p m)
```

In the *Animals* section, the species of an `Animal` is a list of trait genes derived from a `Trait` instance. The `Profile` instances of this module store such lists in their `traits` data members, the only difference being that the sex gene is not one of the genes that define the species of an `Animal`. The `deriveSpecies` function of this module, which has the same name as a similar function in *Animals*, takes a list of trait genes as input and removes the sex gene from the second position.

Inputs:

- complete list of trait genes

```
>deriveSpecies ::  [Float] → [Float]
>deriveSpecies (a:_:cs) = a:cs
```

The `speciesEq` function works similarly to the function in the *Animals* section of the same name, except that it expects its input to be in a different form. Rather than taking `Traits` instances as input, this function takes two lists of trait genes, each having already had their sex genes removed by `deriveSpecies`.

Inputs:

- two lists of trait genes with the sex genes removed

```
>speciesEq ::  [Float] → [Float] → Bool
>speciesEq [] [] = True
>speciesEq (t:ts) (s:ss) = (|t - s| < speciesEqualityConstant) ∧ (speciesEq ts ss)
```

Of interest are groups of animals that are of the same species. Given a `Profile`, we wish to find all other `Profile`'s in the population that describe a member of the same species as the given `Profile` describes. The `fellowMembers` function takes a `Profile` and list of `Profile`'s, and whittles the list down to those

Profile's describing members of the same species as the individual Profile. This is done with a filter call.

Inputs:

- Profile
- list of Profile's for the whole population.

```
>fellowMembers :: Profile → [Profile] → [Profile]
>fellowMembers p ps = filter ((speciesEq s).deriveSpecies.traits) ps
>   where
>      s = deriveSpecies (traits p)
```

Being interested in multiple populations of different species, we now go on to define the groupSpecies function, which works through multiple applications of the fellowMembers function within a helper function named go. The groupSpecies function takes a single list of Profile's as input and returns a list of lists of Profile's, such that every possible grouping of animals of the same species is represented. That is to say that each list within the list of lists contains Profile's whose members are of the same species. If speciesEq is thought of as a relation on the set of animals in the population, then the elements of the list returned by groupSpecies can be thought of as classes of the speciesEq relation. The class for an animal a is defined as $\{b \mid$ speciesEq a b$\}$. However, these are not equivalence classes. The speciesEq relationship is reflexive and symmetric, but not transitive. It is possible for one Profile to be within multiple relational classes. Another way to say this is that the intersection of any two classes in the output list is potentially non-empty.

The fellowMembers function is called with each Profile in the population as the first input and the remainder of the population as the second input. The number of lists generated equals the number of animals in the population. The output set could potentially contain the same list more than once, though strictly speaking, these lists may not be equal in Haskell because their elements could be arranged in different orders.

Inputs:

- list of Profile's for the whole population.

```
>groupSpecies :: [Profile] → [[Profile]]
>groupSpecies ps = go ps (length ps)
>   where
>      go :: [Profile] → Int → [[Profile]]
>      go _ 0 = []
>      go (o:os) n = (o:same):(go (os++[o]) (n-1))
>         where
>            same = fellowMembers o os
```

In order to simplify data collection, functions that combine the many functions defined above are defined. These functions create Report's, which are data instances composed of the results from several of the functions defined above. Different types of Report's hold different data.

A Counts type of Report holds counts of several different types of Animal's and one count of the number of plants. The first three counts are the numbers of total Animal's, living Animal's and dead Animal's. Next is the count of plants. Then there is an ActionData instance, which contains counts of the number of Animal's that performed each possible action on a given time step. The next two data members are labeled male and female, and count the numbers of Animal's of each of the two sexes. Next is an EatingData instance, which contains counts for the numbers of Animal's with each possible diet type. The next two counts are of mature and immature Animal's, followed by Animal's whose mating countdowns are zero and those whose are not.

A Statistics type holds the results of applying a statistical function, such as taking the average, to several lists of living Animal properties and traits. Technically, two different functions are needed if the general type of Num is not general enough to define the function. Often a function that works on a list of Floating types will not work on a list of Integral types, so two functions are required to perform the same operation on different number types. It is possible to use two completely unrelated functions to create one `Statistics` instance, but this is not advised. The first member of a `Statistics` instance comes from applying the statistical function to a list of `Animal`'s energies. The following results come from lists of `Animal` ages, mating countdowns, maximum energies, death points, starting energies, sight ranges, maneuverabilities, maximum movement speeds, lifespans, mutation rates, radii, mating costs, learning rates, maturity ages, mating times, dummy traits and mating recoveries.

```
>data Report
> = Counts {total::Int, live::Int, dead::Int, plantCount::Int,
>      action::ActionData, male::Int, female::Int, eating::EatingData,
>      mature::Int, immature::Int, matingReady::Int, matingRecovering::Int}
> | Statistics {energies::Float, ages::Float, matingCountdowns::Float,
>      maximumEnergies::Float, deathPoints::Float, startingEnergies::Float, sights::Float,
>      maneuverabilities::Float, maximumSpeeds::Float, lifespans::Float,
>      mutationRates::Float, radii::Float, matingCosts::Float, learningRates::Float,
>      maturityAges::Float, dummyTraits::Float, matingRecoveries::Float}
>         deriving Show
```

The `reportCountsA` and `reportCountsR` functions make `Counts` type `Report` instances for the entire world and a single `Region` respectively. The `reportCountsA` function calls the `animalsA`, `plantsA`, `animalActionsA`, `animalSexesA`, `animalDietsA`, `animalMaturitiesA` and `animalMatingCountdownsA` functions to get the data needed to create a `Counts` instance. The `reportCountsR` function calls the corresponding `R` versions of the same functions to produce its `Counts` instance.

Inputs:

- `Moment`

```
>reportCountsA ::  Moment → Report
>reportCountsA mom = Counts t l d p a m f e ma im ready rec
>   where
>      (t,l,d) = animalsA mom
>      p = plantsA mom
>      a = animalActionsA mom
>      (m,f) = animalSexesA mom
>      e = animalDietsA mom
>      (ma,im) = animalMaturitiesA mom
>      (ready,rec) = animalMatingCountdownsA mom
```

Inputs:

- index of the `Region` to take data from
- `Moment`

```
>reportCountsR ::  (Int,Int) → Moment → Report
>reportCountsR pos mom = Counts t l d p a m f e ma im ready rec
>   where
>      (t,l,d) = animalsR pos mom
>      p = plantsR pos mom
>      a = animalActionsR pos mom
>      (m,f) = animalSexesR pos mom
>      e = animalDietsR pos mom
>      (ma,im) = animalMaturitiesR pos mom
>      (ready,rec) = animalMatingCountdownsR pos mom
```

The next two functions make Statistics type instances of Report. Of the data represented in a Statistics instance, the following are properties that can be derived directly from a Profile: energy, age, mating count, maximum energy and death point. All properties are retrieved with either livePropertiesA or livePropertiesR, depending on whether the A or R version of the statistics reporting function is used. The rest of the data are traits derived from the traits chromosome of a Profile, and are therefore retrieved with either traitA or traitR, and the appropriate trait deriving function. The traits represented are: starting energy, sight, maneuverability, maximum movement speed, lifespan, mutation rate, radius, mating cost, learning rate, maturity age, dummy trait and mating recovery.

Two functions are sent to the statistics reporting function as input. These two functions should be essentially the same, except that one takes a list of Int values as input and the other takes a list of Float values as input. These two functions should be two versions of the same statistical function. Each list of properties and traits has one of these two functions applied to it to produce a single Float value. The Int type function is applied to the lists of ages, mating countdowns, lifespans, maturity ages, mating times and mating recoveries. The Float type function is applied to lists of energies, maximum energies, death points, starting energies, sights, maneuverabilities, maximum movement speeds, mutation rates, radii, mating costs, learning rates and dummy trait values.

Inputs:

- an Integral type statistics function
- the function's Floating type equivalent
- Moment

```
>reportStatisticsA ::  ([Int] → Float) → ([Float] → Float) → Moment → Report
>reportStatisticsA i f mom
>   = Statistics en ag mCnt maxE dPnt sEn sight man ms ls mr rad mc lr mat dt mRec
> where
>   en = f (livePropertiesA energy mom)
>   ag = i (livePropertiesA age mom)
>   mCnt = i (livePropertiesA matingCountdown mom)
>   maxE = f (livePropertiesA maxEnergy mom)
>   dPnt = f (livePropertiesA deathPoint mom)
>   sEn = f (traitA deriveStartingEnergy mom)
>   sight = f (traitA deriveSight mom)
>   man = f (traitA deriveManeuverability mom)
>   ms = f (traitA deriveMoveSpeed mom)
>   ls = i (traitA deriveLifespan mom)
>   mr = f (traitA deriveMutationRate mom)
>   rad = f (traitA deriveRadius mom)
>   mc = f (traitA deriveMatingCost mom)
>   lr = f (traitA deriveLearningRate mom)
>   mat = i (traitA deriveMaturity mom)
>   dt = f (traitA deriveDummyTrait mom)
>   mRec = i (traitA deriveMatingRecovery mom)
```

Inputs:

- an Integral type statistics function
- the function's Floating type equivalent
- the index of the Region to take data from
- Moment

```
>reportStatisticsR :: ([Int] → Float) → ([Float] → Float) → (Int,Int) → Moment → Report
>reportStatisticsR i f p mom
>   = Statistics en ag mCnt maxE dPnt sEn sight man ms ls mr rad mc lr mat dt mRec
> where
>   en = f (livePropertiesR energy p mom)
>   ag = i (livePropertiesR age p mom)
>   mCnt = i (livePropertiesR matingCountdown p mom)
>   maxE = f (livePropertiesR maxEnergy p mom)
>   dPnt = f (livePropertiesR deathPoint p mom)
>   sEn = f (traitR deriveStartingEnergy p mom)
>   sight = f (traitR deriveSight p mom)
>   man = f (traitR deriveManeuverability p mom)
>   ms = f (traitR deriveMoveSpeed p mom)
>   ls = i (traitR deriveLifespan p mom)
>   mr = f (traitR deriveMutationRate p mom)
>   rad = f (traitR deriveRadius p mom)
>   mc = f (traitR deriveMatingCost p mom)
>   lr = f (traitR deriveLearningRate p mom)
>   mat = i (traitR deriveMaturity p mom)
>   dt = f (traitR deriveDummyTrait p mom)
>   mRec = i (traitR deriveMatingRecovery p mom)
```

So far there has been an `A` type and `R` type for each data collection function. The `A` type functions collect data from the entire world and the `R` type functions collect data from a single `Region`. One `Region` is often of interest in comparison to other `Region`'s. Such a comparison requires multiple applications of the `R` version of a function. An `R` type function applied to every `Region` in the world can be more useful and meaningful than the `A` version of a function, because the multiple calls provide an image of the entire world, but the individual calls provide images of local areas. In order to facilitate the application of an `R` type function to every `Region` in the world, the `byRegion` function is defined.

All `R` type functions take the index of a `Region` and a `Moment` as the final two inputs. For any `R` type function that has inputs preceding these two inputs, the filling in of preceding inputs creates a new function that takes only a `Region` index and a `Moment` as input. The resulting functions all have the same signature, and the `byRegion` function takes as input a function with this signature. The `byRegion` function takes this input function, which is either an `R` type function or is derived from one, and applies it to every `Region` in a given `Moment` to produce as output a list of `Region` indices coupled with the outputs produced from the input function applied to the corresponding `Region`'s. The traversal of the `BRAMatrix` of `Region`'s is done by the helper function `outAll`, which processes the `BRAMatrix` in row major order.

Inputs:

- an `R` type function or a function derived from one (by filling in initial inputs)
- `Moment`

```
>byRegion :: ((Int,Int) → Moment → a) → Moment → [((Int,Int),a)]
>byRegion f m = outAll (0,0)
>   where
>      outAll (x,y)
>         | y ≡ binGridSizeY = []
>         | x ≡ binGridSizeX = outAll (0,y+1)
>         | otherwise = ((x,y),(f (x,y) m)):(outAll (x+1,y))
```

All data collection functions in this module take a `Moment` as input. A `Moment` is a complex data structure derived from a single output file of the simulation. To gain information about a single time step from the simulation, the output file produced from that time step must be parsed into a `Moment` using the functions at the beginning of this module. This Moment, once created, can be used as input to any of the data gathering functions in this module.

The command function combines these two steps. It allows the user to specify a file to open using the number in the file's name, which is the time step on which the file was created. The readMoment function is used to parse a Moment from this file. The command function also takes a function as input. This function can be any of the data gathering functions in this module as long as all inputs preceding the Moment input are filled in. Once the Moment is parsed, it is sent to this function as input, and the result, assuming a file existed for the desired time step, is returned as output. Because the output was generated using data from a file, the output type is an IO type.

Inputs:

- A type function, a function derived from an A type function (by filling in preceding inputs) or an R type function with all preceding inputs including the Region index filled in
- the time step of the file to open (the number is the suffix of the filename before the extension)

```
>command ::  (Moment → a) → Integer → IO a
>command f n =
>   do
>      m ← readMoment n
>      let
>         result = (maybe (error (‘‘No data for time ’’++(show n)++‘‘.’’))  f m)
>      return result
```

In order to analyze the changes in the population over time, one has to view the results of data collections on several time steps. Given the results from every time step in a given range, one can make graphs of the data across time. The fromTimeRange function performs a given data collection function on every time step between and including two input values. The result is an IO list of the output created for each number in the sequence. The function starts collecting data at the lower bound and increments this value on each recursive step until the lower and upper bounds are equal. Once this base case is reached, the Moment's are parsed and processed as the recursive calls collapse.

Inputs:

- A type function, a function derived from an A type function (by filling in preceding inputs) or an R type function with all preceding inputs including the Region index filled in
- two time steps for the first and last files in the range to check (with the first value at most the last)

```
>fromTimeRange ::  (Moment → a) → Integer → Integer → IO [a]
>fromTimeRange f n m
> | n ≡ m =
>   do
>      mom ← readMoment n
>      return ([maybe (error (‘‘No data for time ’’++(show n)++‘‘.’’))  f mom])
> | otherwise =
>   do
>      rs ← fromTimeRange f (n+1) m
>      mom ← readMoment n
>      let
>         results = (maybe (error (‘‘No data for time ’’++(show n)++‘‘.’’))  f mom):rs
>      return results
```

Although IO type functions can print data to the screen, an IO type value returned by a function is not displayed at all. This value must be caught and dealt with by another IO function for the result to be seen, or perhaps further processed. Two simple and generic functions are defined below to catch such an IO value and deal with it. The `showIO` function will print whatever result it receives to the screen, and the `writeIO` function will write the result it receives to a file. The filename is given as input. In both cases there must exist a `Show` instance for the result in order for it to be displayed.

Inputs:

- an `IO` type value with a `Show` instance

```
>showIO :: Show a ⇒ IO a → IO ()
>showIO x = do
>    y ← x
>    print y
```

Inputs:

- file to write output to
- an `IO` type value with a `Show` instance

```
>writeIO :: Show a ⇒ String → IO a → IO ()
>writeIO name x = do
>    y ← x
>    writeFile name (show y)
```

A.4 Report

The functions of the this module complement those of the *Data Analysis* module by summarizing the data those functions produce into text file reports. These files contain information sorted into columns. This format makes it easy to import the data into Microsoft Excel, which is used to create graphs and charts of the output. Excel's capabilities make coding comparable features into a Haskell module unnecessary, which is especially convenient since Haskell has no standardized graphics library.

As with the *Data Analysis* module, some of the functions below deal with `Region`'s and others deal with the entire world. The functions in this module can also be classified in terms of whether they generate reports from a single `Moment` or from several `Moment`'s in a given range of time. The last two letters of these functions reveal their classification. The first of these can be either `R` or `A`, which means the function operates on one `Region` or on all `Region`'s respectively. The second letter can be `M` or `T`, meaning that the function operates on one `Moment` or on several `Moment`'s in a time range respectively.

```
>module Report
> (arbitrate, excelChunk, selectMoments,
> plantPositionsRM, plantPositionsAM,
> livePositionsRM, livePositionsAM, deadPositionsRM, deadPositionsAM,
> reportCountsRT, reportCountsAT,
> reportAveragesAT, reportAveragesRT,
> reportStdDevsAT, reportStdDevsRT) where
```

Reports are generated using functions from the *Data Analysis* module. Some helper functions from *Standard Code Extensions* are also used, as are several values from *Constants*, and some file handling functions from `IO`.

```
>import Analyze -- Data Analysis
>import StandardExts -- Standard Code Extensions
>import Constants -- Constants
>import IO
```

Any function that generates a report from many `Moment`'s in a given time range must open all of the output files from the simulation corresponding to that time range. These files are accessed by the *Data Analysis* module's `fromTimeRange` function and the data taken from them is rearranged into columns by one of this module's many organizing functions. However, there is a problem that arises when accessing so many files. There is a limit on how many files can be open at one time, and Haskell's lazy evaluation and functional nature makes assuring file closure difficult. As a result, all files opened by `fromTimeRange` stay open for the duration of that function's execution.

The `arbitrate` function addresses this problem by making multiple separate calls to the report generating function, which in turn assures that multiple separate calls to `fromTimeRange` are made. For every call to the reporting function, data from a portion of the time range is collected and written to a file using append mode. This allows each successive call to add more data to the same file until the report is completed. The maximum number of files examined at one time is set by `arbitrateSize`, which can be any number less than the system's open file limit.

```
>arbitrateSize = 800
```

Besides containing data taken from the simulation's output files, a report also has headings for each column of data. This means that the initial call to the reporting function must be different from all subsequent calls. This difference is controlled by a Boolean value that all data reporting functions take as input. If the value is `True`, then the headings are written to the file. If the value is `False`, then only the data is written. The `arbitrate` makes sure that the first call to the data reporting function is made with a value of `True`, and that subsequent calls are made with a value of `False`.

An additional complication arises from the use of Microsoft Excel to graph data from the reports. Excel suffers from the limitation of being unable to plot more than 32,000 points on a 2D graph. This means that a report with more than this number of entries could not be fully analyzed, and would have to be broken into parts. Since having such a large report would result in extra work, the `excelChunk` function is used to instead create several smaller reports in place of one. The number of data entries allowed in a single file is determined by `excelSize`, which can be at most 32,000 and greater than `arbitrateSize`.

```
>excelSize = 10000
```

The basic function signature of functions that report data from a range of `Moment`'s includes a Boolean indicator for whether the headings should be written, the filename to output to (including file extension), as well as the first and last time steps in the range of `Moment`'s to collect data from. The `arbitrate` function takes such a function as input along with the aforementioned filename and time range boundaries. Only if the difference between the start and end `Moment`'s is greater than `arbitrateSize` does recursion occur. Otherwise the data reporting function is called a single time and all data is written to it at once.

When recursion occurs, it is the helper function `cycle` that is recursive. It has a start case, a recursive case and an end case. Every call to `cycle`, except for the last one, processes `arbitrateSize` number of data files. The end case processes the remaining data files. The start case differs from the others in that it sends a value of `True` to the data reporting function so that the headings will be written to the top of the output file. The other two cases send a value of `False` so that the data they produce is seamlessly appended to all previous output.

Inputs:

- data reporting function of the `T` type (with inputs preceding the Boolean indicator filled in)
- filename to write to (with extension)
- first time step to collect data on
- last time step to collect data on

```
>arbitrate ::  (Bool → String → Integer → Integer → IO ())
>       → String → Integer → Integer → IO ()
>arbitrate f filename start stop
> | (stop - start) > arbitrateSize =
>    do
>       writeFile filename ‘‘’’
>       cycle start (start + arbitrateSize)
> | otherwise =
>    do
>       writeFile filename ‘‘’’
>       f True filename start stop

> where
>    cycle ::  Integer → Integer → IO ()
>    cycle n m
>       | n ≡ start =
>          do
>             f True filename n m
>             cycle (m + 1) (m + arbitrateSize)
>       | m > stop =
>          do
>             f False filename n stop
>       | otherwise =
>          do
>             f False filename n m
>             cycle (m + 1) (m + arbitrateSize)
```

The mechanics of the `excelChunk` function are similar to those of `arbitrate`, except that `excelChunk` works with larger portions of data and creates a new file for each portion. Each of these files shares a common name followed by the range of values it reports on: the lower and upper bounds separated by a dash. The common name is sent to the function as input. When a file is written, this name is appended to the time range and the ".txt" extension. The extension is supplied by the function itself and is not part of the input.

The lower and upper time steps of the range to report on are sent to the function. If the difference between these two values is less than `excelSize`, then only a single call to `arbitrate` is made and only a single file is produced. Otherwise the helper function `cycle` recursively collects data and creates files. The `cycle` function has a recursive case and an end case only. The only difference between the two besides the recursive call is that the end case only collects data from the remaining data files and the other cases all collect data from `excelSize` number of data files.

Inputs:

- data reporting function of the `T` type (with inputs preceding the Boolean indicator filled in)
- common filename (a prefix of each output file. No extension needed)
- first time step to collect data on
- last time step to collect data on

```
>excelChunk ::  (Bool → String → Integer → Integer → IO ())
>       → String → Integer → Integer → IO ()
>excelChunk f filename start stop
> | (stop - start) > excelSize =
>    do
>       cycle start (start + excelSize)
> | otherwise =
>    do
>       let
>          newName = filename ++ (show start) ++ ‘‘-’’ ++ (show stop) ++ ‘‘.txt’’

>       arbitrate f newName start stop

> where
>    cycle ::  Integer → Integer → IO ()
>    cycle n m
>       | m > stop =
>          arbitrate f (filename++(show n)++‘‘-’’++(show stop)++‘‘.txt’’) n stop
>       | otherwise =
>          do
>             arbitrate f cycleName n m
>             cycle (m + 1) (m + excelSize)
>          where
>             cycleName = filename ++ (show n) ++ ‘‘-’’ ++ (show m) ++ ‘‘.txt’’
```

If a function only handles data from a single `Moment`, then the only way to get a sense of what this data means throughout time is to execute the function repeatedly on all `Moment`'s in a given time range. The `selectMoments` function takes such `M` type functions as input and creates a file of output for each Moment that it analyzes. A common name is given to the function as input, and the final name of each report file is this common name concatenated to the time step it corresponds to followed by the ".txt" file extension. The starting and ending time steps are provided as inputs and recursive calls create report files as a counter goes from the start value to the end value, at which point recursion stops.

Inputs:

- an `M` type function (which takes a filename and a time step value as input)
- common filename prefix (no file extension)
- first time step to collect data on
- last time step to collect data on

```
>selectMoments ::  (String → Integer → IO ())
>    → String → Integer → Integer → IO ()
>selectMoments f filename start stop
> | start ≡ stop =
>    do
>       f newName start
> | otherwise =
>    do
>       f newName start
>       selectMoments f filename (start + 1) stop

> where
>    newName = filename ++ (show start) ++ ‘‘.txt’’
```

The first two data reporting functions in this module return lists of plant positions in a `Region` and the entire world respectively. The structure of the two functions is identical. The only difference between them is that the `R` version of this function uses the `R` version of the function that gathers the plant position data and the `A` version of this function uses the `A` version of the plant positions function. Naturally the `R` version also has the index of a `Region` as input. Both versions take a filename and a time step as input. The functions work by first gathering information from the file of the particular time step, which is in the format of a list of 2-tuples (for the x and y coordinates). Then an output string is created starting with the headings "X Coord" and "Y Coord". The gathered data is organized into two columns by the `organize2` function and added to this string, which is written to the file with the `hPutStr` command after the file has been opened with the `openFile` command.

Inputs:

- `Region` index
- filename to write to (including extension)
- time step to gather data from

```
>plantPositionsRM ::  (Int,Int) → String → Integer → IO ()
>plantPositionsRM i filename time =
> do
>    result ← command (plantPositionsR i) time

>    let
>       headings = ‘‘X Coord\tY Coord\n’’
>       output = headings++(organize2 result)

>    h ← openFile filename AppendMode
>    hPutStr h output
>    hClose h
```

Inputs:

- filename to write data to (including extension)
- the time step to gather data from

```
>plantPositionsAM ::  String → Integer → IO ()
>plantPositionsAM filename time =
> do
>    result ← command plantPositionsA time

>    let
>       headings = ‘‘X Coord\tY Coord\n’’
>       output = headings++(organize2 result)

>    h ← openFile filename AppendMode
>    hPutStr h output
>    hClose h
```

The `organize2` function takes a list of 2-tuples and organizes the data into a string such that it appears in two columns when printed. The contents of the tuple must have `Show` instances for this to work. For each tuple in the list, the two tuple values are added to a string with a tab between them. A newline is appended after this and then a recursive call to `organize2`, which adds the data from the remaining tuples. When the list of tuples is empty the function is finished.

Inputs:

- list of 2-tuples of two showable types

```
>organize2 ::  (Show a, Show b) ⇒ [(a,b)] → String
>organize2 [] = ‘‘’’
>organize2 ((a,b):rs) = (show a)++‘‘\t’’++(show b)++‘‘\n’’++(organize2 rs)
```

The next functions provide the positions of animals. The ID numbers of the animals are also provided so that there is a way to track the movements of individual animals through time. There are four variations of this function: living animals in one `Region`, living animals in the entire world, dead animals in one `Region`, and dead animals in the entire world. This makes for two `R` type functions and two `A` type functions. The `R` type functions are the simpler of the two types, because they gather and report data without changing it. The `A` type functions change the position data before reporting it because an animal's position is stored relative to the `Region` it is in. In order to give an accurate world view, the positions of the animals are offset according to the `Region` they are in.

The function that offsets these position values is `collectPositions`. The world is organized into adjacent `Region`'s of equal size. Therefore an animal's global x-coordinate is its position within its `Region` added to the product of that `Region`'s x-coordinate (in the grid of `Region`'s) and the width of the `Region`. An animal's global y-coordinate is similarly calculated using height instead of width and the `Region`'s y-coordinate instead of its x-coordinate. The `collectPositions` function adjusts all animal positions in this manner after collecting them using the appropriate function: either `livePropertiesR` or `deadPropertiesR`. The function works by recursively calling itself to check every `Region` in the world, but the initial call should be done with `Region` index (0,0). The final input is a list accumulator which should start out empty. It collects all the results which are returned at the end of the function's execution.

Inputs:

- time step to collect data from
- either the `livePropertiesR` or `deadPropertiesR` function
- starting index (0,0)
- an empty list to serve as an accumulator

```
>collectPositions ::  Integer → ((Profile → (Integer,Float,Float))
>     → (Int,Int) → Moment → [(Integer,Float,Float)])
>     → (Int,Int) → [(Integer,Float,Float)] → IO [(Integer,Float,Float)]
>collectPositions time f (x,y) vs
> | x ≡ binGridSizeX = collectPositions time f (0, y + 1) vs
> | y ≡ binGridSizeY = do return vs
> | otherwise =
>    do
>       r ← command
>          (f (λ p → (idN p, (fst.position) p, (snd.position) p)) (x,y)) time

>       let
>          shifted = map (λ (id,x1,y1) →
>             (id, x1 + ((convert x) × binWidth), y1 + ((convert y) × binHeight))) r

>       collectPositions time f (x + 1,y) (shifted++vs)
```

The live and dead type functions are only different in that one deals with living animals and the other deals with dead animals. Each of these functions has an `A` and `R` version. The `A` versions are only slightly different from the `R` versions in that they use `collectPositions` to collect data instead of directly calling on the appropriate data collection functions themselves.

Both of the `R` type functions make use of the `positionsRM` function below to collect data. The function sent as input, `livePropertiesR` or `deadPropertiesR`, determines whether information on dead or living animals is returned. The structure of the function is quite similar to `plantPositionsRM` above, except that ID numbers are returned in addition to x and y coordinates. This additional information means that the `organize3` function (defined below) is used instead of the `organize2` function to arrange the data.

Inputs:

- either the `livePropertiesR` or `deadPropertiesR` function
- index of the Region to gather data from
- name of the output file (with file extension)
- time step to gather data from

```
>positionsRM ::  ((Profile → (Integer,Float,Float)) → (Int,Int) → Moment
>    → [(Integer,Float,Float)]) → (Int,Int) → String → Integer → IO ()
>positionsRM f i filename time =
> do
>    result ← command (f (λ p → (idN p, (fst.position) p, (snd.position) p)) i) time

>    let
>       headings = ‘‘ID’s\tX Coord\tY Coord\n’’
>       output = headings++(organize3 result)

>    h ← openFile filename AppendMode
>    hPutStr h output
>    hClose h
```

Both `A` type functions use the `positionsAM` function. It organizes data the same as in `positionsRM`, but it uses `collectPositions` to gather its data.

Inputs:

- either the `livePropertiesR` or `deadPropertiesR` function
- name of the output file (with file extension)
- the time step to gather data from

```
>positionsAM ::  ((Profile → (Integer,Float,Float)) → (Int,Int) → Moment
>       → [(Integer,Float,Float)]) → String → Integer → IO ()
>positionsAM f filename time =
> do
>    result ← collectPositions time f (0,0) []

>    let
>       headings = ‘‘ID’s\tX Coord\tY Coord\n’’
>       output = headings++(organize3 result)

>    h ← openFile filename AppendMode
>    hPutStr h output
>    hClose h
```

With the above functions defined, it is easy to define `livePositionsRM`, `livePositionsAM`, `deadPositionsRM` and `deadPositionsAM`. The `R` and `A` type functions call on the `positionsRM` and `positionsAM` respectively.

The live and dead type functions use `livePropertiesR` and `deadPropertiesR` respectively.

Inputs:

- index of the Region to gather data from
- name of the output file (with file extension)
- the time step to gather data from

```
>livePositionsRM :: (Int,Int) → String → Integer → IO ()
>livePositionsRM = positionsRM livePropertiesR
```

Inputs:

- name of the output file (with file extension)
- the time step to gather data from

```
>livePositionsAM :: String → Integer → IO ()
>livePositionsAM = positionsAM livePropertiesR
```

Inputs:

- index of the Region to gather data from
- name of the output file (with file extension)
- time step to gather data from

```
>deadPositionsRM :: (Int,Int) → String → Integer → IO ()
>deadPositionsRM = positionsRM deadPropertiesR
```

Inputs:

- name of the output file (with file extension)
- time step to gather data from

```
>deadPositionsAM :: String → Integer → IO ()
>deadPositionsAM = positionsAM deadPropertiesR
```

The `organize3` function is essentially identically to the `organize2` function except that it takes 3-tuples instead of 2-tuples as input, and as a result returns a string of three columns instead of two.

Inputs:

- list of 3-tuples of three showable types

```
>organize3 :: (Show a,Show b,Show c) ⇒ [(a,b,c)] → String
>organize3 [] = ‘‘’’
>organize3 ((a,b,c):rs)
> = (show a)++‘‘\t’’++(show b)++‘‘\t’’++(show c)++‘‘\n’’++(organize3 rs)
```

The `organizeCounts` function is yet another function used to organize data into a `String` of columns. It is specifically designed to organize a list of `Counts` instances of the `Report` data type into columns for display. Though the function looks frightening, it is simply a series of concatenated `show` calls on the data members of a `Counts` instance.

Inputs:

- list of Report instances of the Counts type

```
>organizeCounts ::  [Report] → String
>organizeCounts [] = ‘‘’’
>organizeCounts (r:rs) =
> (show (total r))++‘‘\t’’++(show (live r))++‘‘\t’’++(show (dead r))
> ++‘‘\t’’++(show (plantCount r))++‘‘\t’’++(show (passedAway a))
> ++‘‘\t’’++(show (starved a))++‘‘\t’’++(show (rotted a))
> ++‘‘\t’’++(show (eaten a))++‘‘\t’’++(show (born a))
> ++‘‘\t’’++(show (nothing a))++‘‘\t’’++(show (mated a))
> ++‘‘\t’’++(show (atePlant a))++‘‘\t’’++(show (predated a))
> ++‘‘\t’’++(show (cannibalized a))++‘‘\t’’++(show (ateCarrion a))
> ++‘‘\t’’++(show (failedMating a))++‘‘\t’’++(show (male r))
> ++‘‘\t’’++(show (female r))++‘‘\t’’++(show (none e))
> ++‘‘\t’’++(show (herb e))++‘‘\t’’++(show (carn e))
> ++‘‘\t’’++(show (cann e))++‘‘\t’’++(show (carr e))
> ++‘‘\t’’++(show (herbCarn e))++‘‘\t’’++(show (herbCann e))
> ++‘‘\t’’++(show (herbCarr e))++‘‘\t’’++(show (carnCann e))
> ++‘‘\t’’++(show (carnCarr e))++‘‘\t’’++(show (cannCarr e))
> ++‘‘\t’’++(show (herbCarnCann e))++‘‘\t’’++(show (herbCarnCarr e))
> ++‘‘\t’’++(show (herbCannCarr e))++‘‘\t’’++(show (carnCannCarr e))
> ++‘‘\t’’++(show (herbCarnCannCarr e))++‘‘\t’’++(show (mature r))
> ++‘‘\t’’++(show (immature r))++‘‘\t’’++(show (matingReady r))
> ++‘‘\t’’++(show (matingRecovering r))++‘‘\n’’++(organizeCounts rs)
> where
>    a = action r
>    e = eating r
```

The organizeStatistics function works the same as the organizeCounts function, except that it is applied to a list of Statistics type instances of the Report data type.

Inputs:

- list of Report instances of the Statistics type

```
>organizeStatistics ::  [Report] → String
>organizeStatistics [] = ‘‘’’
>organizeStatistics (a:as) =
> (show (energies a))++‘‘\t’’++(show (ages a))++‘‘\t’’++(show (matingCountdowns a))
> ++‘‘\t’’++(show (maximumEnergies a))++‘‘\t’’++(show (deathPoints a))
> ++‘‘\t’’++(show (startingEnergies a))++‘‘\t’’++(show (sights a))
> ++‘‘\t’’++(show (maneuverabilities a))++‘‘\t’’++(show (maximumSpeeds a))
> ++‘‘\t’’++(show (lifespans a))++‘‘\t’’++(show (mutationRates a))
> ++‘‘\t’’++(show (radii a))++‘‘\t’’++(show (matingCosts a))
> ++‘‘\t’’++(show (learningRates a))++‘‘\t’’++(show (maturityAges a))
> ++‘‘\t’’++(show (dummyTraits a))++‘‘\t’’++(show (matingRecoveries a))
> ++‘‘\n’’++(organizeStatistics as)
```

Having defined these two organizing functions, we now define reporting functions that utilize them. First are the Counts reporting functions reportCountsRT and reportCountsAT. These functions use the reportCountsR and reportCountsA functions respectively to create Counts type instances of the Report data type. The organizeCounts function is then used to organize the resulting data into columns, which are printed to an output file.

Inputs:

- index of the Region to gather data from
- Boolean indicator for whether or not to append the headings
- name of the output file (with file extension)
- starting time step to gather data from
- final time step to gather data from

```
>reportCountsRT ::  (Int,Int) → Bool → String → Integer → Integer → IO ()
>reportCountsRT p b filename start stop =
> do
>    result ← fromTimeRange (reportCountsR p) start stop

>    let
>       headings = if b then ‘‘Total\tLive\tDead\tPlants\tPassed Away\tStarved\tRotted’’
>          ++‘‘\tEaten\tBorn\tNothing\tMated\tAte Plant\tPredated\tCannibalized\tAte Carrion’’
>          ++‘‘\tFailed Mating\tMale\tFemale\tnone\therb\tcarn\tcann\tcarr’’
>          ++‘‘\therbCarn\therbCann\therbCarr\tcarnCann\tcarnCarr\tcannCarr\therbCarnCann’’
>          ++‘‘\therbCarnCarr\therbCannCarr\tcarnCannCarr\therbCarnCannCarr\tMature’’
>          ++‘‘\tNot Mature\tReady\tRecovering\n’’ else ‘‘’’
>       output = headings++(organizeCounts result)

>    h ← openFile filename AppendMode
>    hPutStr h output
>    hClose h
```

Inputs:

- Boolean indicator for whether or not to append the headings
- name of the output file (with file extension)
- starting time step to gather data from
- final time step to gather data from

```
>reportCountsAT ::  Bool → String → Integer → Integer → IO ()
>reportCountsAT b filename start stop =
> do
>    result ← fromTimeRange reportCountsA start stop

>    let
>       headings = if b then ‘‘Total\tLive\tDead\tPlants\tPassed Away\tStarved\tRotted’’
>          ++‘‘\tEaten\tBorn\tNothing\tMated\tAte Plant\tPredated\tCannibalized\tAte Carrion’’
>          ++‘‘\tFailed Mating\tMale\tFemale\tnone\therb\tcarn\tcann\tcarr’’
>          ++‘‘\therbCarn\therbCann\therbCarr\tcarnCann\tcarnCarr\tcannCarr\therbCarnCann’’
>          ++‘‘\therbCarnCarr\therbCannCarr\tcarnCannCarr\therbCarnCannCarr\tMature’’
>          ++‘‘\tNot Mature\tReady\tRecovering\n’’ else ‘‘’’
>       output = headings++(organizeCounts result)

>    h ← openFile filename AppendMode
>    hPutStr h output
>    hClose h
```

For `Statistics` type instances of `Report` there are two pairs of reporting functions: one for averages and one for standard deviations. The `R` type functions `reportAveragesRT` and `reportStdDevsRT` both use `reportStatisticsR` to create a list of `Report`'s, and the `A` type functions `reportAveragesAT` and `reportStdDevsAT` both use `reportStatisticsA` to do the same. The average reporting functions both use `averageIntegral` and `averageFloating` from the *Statistics* section for gathering data, and the standard deviation reporting functions both use `standardDeviationIntegral` and `standardDeviationFloating`, also from *Statistics*, for gathering data. In any case, the data that is gathered is organized by the `organizeStatistics` function and then printed to an output file.

Inputs:

- index of the Region to gather data from
- Boolean indicator for whether or not to append the headings
- name of the output file (with file extension)
- starting time step to gather data from
- final time step to gather data from

```
>reportAveragesRT ::  (Int,Int) → Bool → String → Integer → Integer → IO ()
>reportAveragesRT p b filename start stop =
> do
>    r ← fromTimeRange (reportStatisticsR averageIntegral averageFloating p) start stop
>    let
>       headings = if b then ''avEnergy\tavAge\tavMating Countdown\tavMax Energy''
>          ++''\tavDeath Point\tavStart Energy\tavSight\tavManeuverability''
>          ++''\tavSpeed\tavLifespan\tavMutation\tavRadius''
>          ++''\tavMate Cost\tavLearning\tavMaturity\tavDummy''
>          ++''\tavMate Recovery\n'' else ''''
>       output = headings++(organizeStatistics r)

>    h ← openFile filename AppendMode
>    hPutStr h output
>    hClose h
```

Inputs:

- Boolean indicator for whether or not to append the headings
- name of the output file (with file extension)
- starting time step to gather data from
- final time step to gather data from

```
>reportAveragesAT ::  Bool → String → Integer → Integer → IO ()
>reportAveragesAT b filename start stop =
> do
>    r ← fromTimeRange (reportStatisticsA averageIntegral averageFloating) start stop
>    let
>       headings = if b then ''avEnergy\tavAge\tavMating Countdown\tavMax Energy''
>          ++''\tavDeath Point\tavStart Energy\tavSight\tavManeuverability''
>          ++''\tavSpeed\tavLifespan\tavMutation\tavRadius''
>          ++''\tavMate Cost\tavLearning\tavMaturity\tavDummy''
>          ++''\tavMate Recovery\n'' else ''''
>       output = headings++(organizeStatistics r)
>    h ← openFile filename AppendMode
>    hPutStr h output
>    hClose h
```

Inputs:

- index of the Region to gather data from
- Boolean indicator for whether or not to append the headings
- name of the output file (with file extension)
- starting time step to gather data from
- final time step to gather data from

```
>reportStdDevsRT :: (Int,Int) → Bool → String → Integer → Integer → IO ()
>reportStdDevsRT p b filename start stop =
> do
>    r ← fromTimeRange (reportStatisticsR
>       standardDeviationIntegral standardDeviationFloating p) start stop

>    let
>       headings = if b then ‘‘sdEnergy\tsdAge\tsdMating Countdown\tsdMax Energy’’
>          ++‘‘\tsdDeath Point\tsdStart Energy\tsdSight\tsdManeuverability\tsdSpeed’’
>          ++‘‘\tsdLifespan\tsdMutation\tsdRadius\tsdMate Cost\tsdLearning’’
>          ++‘‘\tsdMaturity\tsdDummy\tsdMate Recovery\n’’ else ‘‘’’
>       output = headings++(organizeStatistics r)

>    h ← openFile filename AppendMode
>    hPutStr h output
>    hClose h
```

Inputs:

- Boolean indicator for whether or not to append the headings
- name of the output file (with file extension)
- starting time step to gather data from
- final time step to gather data from

```
>reportStdDevsAT :: Bool → String → Integer → Integer → IO ()
>reportStdDevsAT b filename start stop =
> do
>    r ← fromTimeRange (reportStatisticsA
>       standardDeviationIntegral standardDeviationFloating) start stop

>    let
>       headings = if b then ‘‘sdEnergy\tsdAge\tsdMating Countdown\tsdMax Energy’’
>          ++‘‘\tsdDeath Point\tsdStart Energy\tsdSight\tsdManeuverability\tsdSpeed’’
>          ++‘‘\tsdLifespan\tsdMutation\tsdRadius\tsdMate Cost\tsdLearning’’
>          ++‘‘\tsdMaturity\tsdDummy\tsdMate Recovery\n’’ else ‘‘’’
>       output = headings++(organizeStatistics r)

>    h ← openFile filename AppendMode
>    hPutStr h output
>    hClose h
```

Bibliography

[1] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 1999.

[2] Donald Hebb. *The Organization of Behavior: A Neuropsychological Theory.* Wiley, 1949.

[3] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[4] Donald Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley Publishing Company, 1969.

[5] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[6] Christopher G. Langton. Artificial Life. In Christopher G. Langton, editor, *Artificial Life*, pages 1–47. Addison-Wesley Publishing Company, 1989.

[7] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[8] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2003.

[9] Simon Thompson. *The Craft of Functional Programming.* Addison Wesley Longman Limited, 1999.

[10] Christopher R. Ward, Fernand Gobet, and Graham Kendall. Evolving Collective Behavior in an Artificial Ecology. *Artificial Life*, 7(2):191–209, 2001.