Amygdala

A spiking neural network library

Version 0.2, April 2002

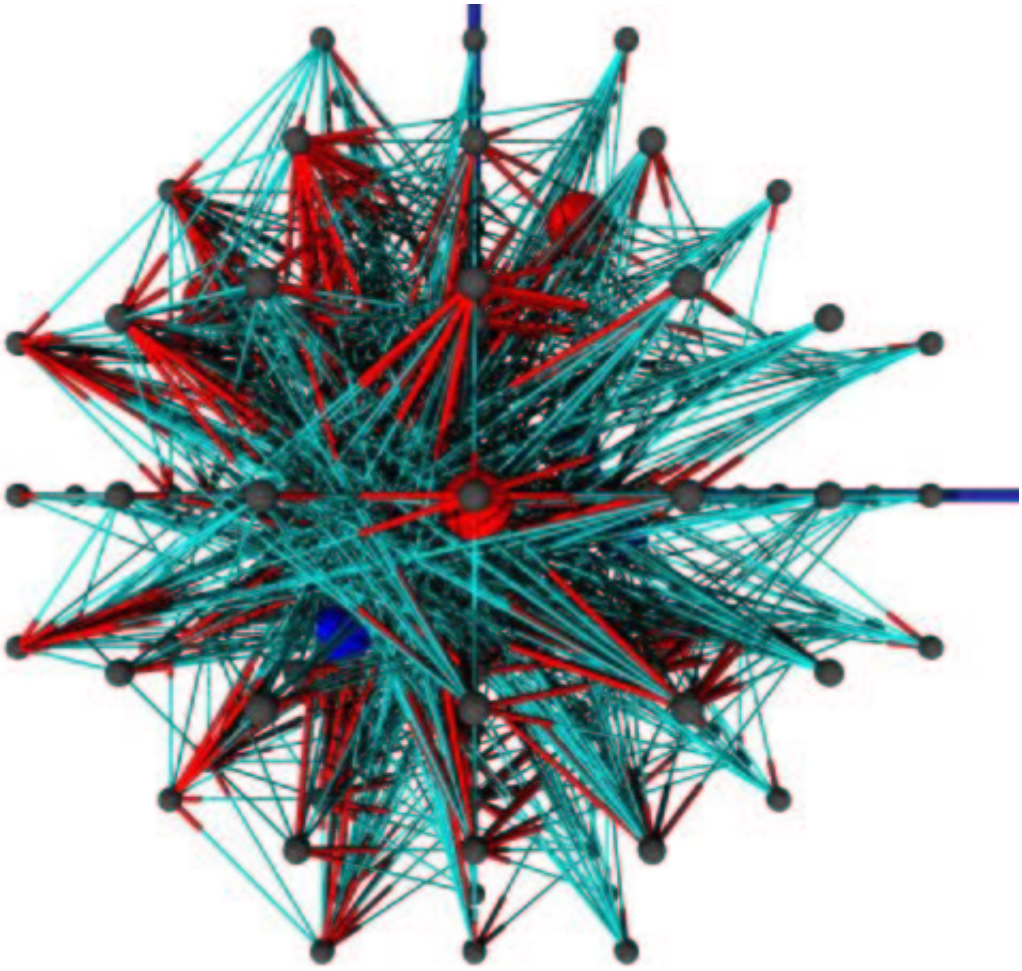


Table of Contents

1. Introduction

Amygdala performs computation by simulating biologically realistic neurons. Neurons of higher animals communicate with each other through a spike protocol. The timing of these spikes holds the information the neurons are exchanging.

Since Amygdala mimics the model mother nature provided it is not surprising that Amygdala can be used for tasks animals are strong at. This includes all types of spatio-temporal pattern recognition tasks such as finding features in a video or speech recognition.

Another important application of Amygdala is the complete simulation of animals. Such simulations can yield a proof, at least strong evidence, that we have the right model of biological neurons. As a prospect for the years ahead we might be able to upload animals including their individual traits, eventually humans.

2. Installation

2.1 How to obtain Amygdala

The Amygdala homepage is http://amygdala.sourceforge.net. Amygdala can be downloaded from http://sourceforge.net/projects/amygdala.

2.2 Requirements

Amygdala is developed on Linux/Intel. It has been compiled and briefly tested on Solaris8/Sparc/gcc-2.95. It should be easy to port it to other platforms as long as Apache/Xerces has been ported to it and it supports Posix.1, Posix threads and the STL.

It might be worthwile to try Amygdala with Intels C++ aka Kai C++ compiler because this would result in a significant performance gain. Most other compilers won't work. Compilers that differ significantly from gcc such as IBMs xlC will probably never be supported.

2.3 Compilation and installation

In order to compile and install Amygdala on your system, type the following in the base directory of the Amygdala distribution:

```
% ./configure
% make
% make install
```

Since Amygdala uses `autoconf` you should have not trouble compiling it. Should you run into problems please report them to the Amygdala mailing list

amygdala-development@lists.sourceforge.net

3. Basic usage of Amygdala

This is a simple guide to programming with the Amygdala library. It is not meant to be exhaustive or complete, but to provide the minimum needed to write a simple program using the library. Complete documentation will be available at a later date.

3.1. Building the network

There are several methods available to users of the library to build networks programatically. In most cases, they will want to build the network in code, and then save it off to a file (either immediately, or after training for a while). At the lowest level of the API, only two objects are required to construct an Amygdala network: a Network object and a member of the Neuron family of classes. Neurons are added to a Network by calling Network::AddNeuron(LayerType, Neuron*) and Neurons are connected together by calling Network::ConnectNeurons(Neuron*, Neuron*, float), which will also set the initial weight of the connection.

```
Network* net = new Network;
AlphaNeuron* nrn;

// Create an AlphaNeuron with neuron ID 1
nrn = new AlphaNeuron(1);

// Set the synaptic and membrane time constants
// for the neuron
nrn->SetTimeConstants(2.0, 10.0);

// Add the neuron to the network
net->AddNeuron(INPUTLAYER, nrn);

// Set up neuron 2
nrn = new AlphaNeuron(2);
nrn->SetTimeConstants(2.0, 10.0);
net->AddNeuron(OUTPUTLAYER, nrn);

// Connect neuron 1 to neuron 2 with an initial
// weight of 0.25.  Note that weights are normalized
// in Amygdala to be in the range -1.0 -> +1.0.
net->ConnectNeurons(1, 2, 0.25);
```

It should be noted that INPUTLAYER neurons are not able to accept connections from other neurons. If a network is designed that requires feedback into the input layer, or if a single-layer network is needed, the INPUTLAYER neurons can be treated as pseudo-neurons. In other words, the INPUTLAYER neurons are used to

feed signals into a second, functional input layer made up of HIDDENLAYER or OUTPUTLAYER neurons. A network that is designed to have a single layer will be implemented as a two layer network with each input neuron connecting to a single output neuron with a weight of 1.0. This allows input to be fed into the network while also allowing connections to be formed within the single functional layer.

Layering is also supported in the API, and building layers is generally easier than adding neurons individually. Users can call Layer::AddNeuron() to add neurons to a layer, or they can call one of the NetworkLoader::BuildLayer() functions, which create a layer with neurons and return a pointer. There are two BuildLayer functions right now. More will be added later on to give users more control over how the layers are constructed. The next code snippet demonstrates the construction of a simple two layer network which is saved to an XML file.

```
AmIdInt startId=0;
LayerConstants layerConst;
Network* net = new Network();
NetworkLoader<BasicNeuron>* netLoad =
    new NetworkLoader<BasicNeuron>(net);
Layer* layer1;
Layer* layer2;
GaussConnectType conParms;

// set up the layer constants
layerConst.type = INPUTLAYER;
layerConst.layerId = 1;
layerConst.learningConst = 1e-3;
layerConst.membraneTimeConst = 10.0;
layerConst.synapticTimeConst = 2.0;
layerConst.restPtnl = 0.0;
layerConst.thresholdPtnl = 5.0;
startId = testNet->MaxNeuronId() + 1;

// build the first layer
// this will build a layer containing 2 neurons
layer1 = netLoad->BuildLayer(2, startId, layerConst);
layer1->LayerName("Layer 1");

// build the second layer
layerConst.layerId = 2;
layerConst.type = OUTPUTLAYER;
layer2 = netLoad->BuildLayer(2,
                             startId + 2,
                             layerConst);
layer2->LayerName("Layer 2");

// set up the connection parameters using a gaussian
// distribution to assign initial weight values
conParms.meanWeight = 0.5;    // average weight
conParms.stdDev = 0.15;       // standard deviation
conParms.pctConnect = 100.0;  // percentage of
                              // neurons to connect

// make 10 percent of the connections inhibitory
layer1->SetPercentInhibitory(10.0);

// connect layer1 to layer2
layer1->ConnectLayers(layer2, conParms);

netLoad->SaveXML("LayerSave.xml");

delete testNet;
delete netLoad;
```

An alternative to building the network programatically is to load a pre-existing network definition file with NetworkLoader. This is an XML file following the format in Amygdala.dtd (distributed with the library), so simple networks can be written manually. To load a file into the network:

```
    Network* net = new Network();
    NetworkLoader<BasicNeuron>* netLoad = new
        NetworkLoader<BasicNeuron>(net);
    netLoad->LoadXML("foo.xml");
```

3.2. Input and Output

Amygdala provides a simple interface for feeding input into the network and reporting on output. A pair of abstract base classes, SpikeInput and SpikeOutput, provide the interface framework and users of the library are able to extend these to suit their needs. The SimpleSpikeInput and SimpleSpikeOutput classes are also provided as ready-made mechanisms to handle I/O functions.

The network can be instructed to use an instance of a SpikeInput class by passing a pointer to a SpikeInput object through through the Network::SetSpikeInput(SpikeInput*) function. Neuron's will use a SpikeOutput class after the static Neuron::SetSpikeOutput(SpikeOutput*) function is called. The SpimpleSpikeInput and SimpleSpikeOutput classes are instantiated by automatically by Amygdala, so nothing has to be done by the developer to before using these classes.

3.2.1 SpikeInput

3.2.1.1 Streaming input

If streaming input has been turned on for Network (Network::StreamingInput(true)), then the SpikeInput::ReadInputBuffer() function will be called from Network::Run() whenever Network's input buffer is empty. The derived SpikeInput class will be responsible for acquiring new input and scheduling input spikes through the Network::ScheduleNEvent() interface. ScheduleNEvent() has three arguments: an event type, event time (spike time), and either a pointer to a neuron or a neuron id. All events scheduled from a SpikeInput class should have an event type of INPUTSPIKE.

```
FooSpikeInput::ReadInputBuffer()
{
    // Example implementation of a ReadInputBuffer() function

    unsigned int i;
```

```
    // Fill the inputSpike vector with input from some
    // user-defined source
    ... // user code

    // sort the vector before scheduling input.
    sort(inputSpike.begin(), inputSpike.end());

    for (i=0; i<inputSpike.size(); i++) {
        net->ScheduleNEvent(INPUTSPIKE,
                            inputSpike[i].spikeTime,
                            inputSpike[i].neuronId);
    }

    inputSpike.clear();
}
```

The above example makes use of the protected vector<InSpike> inputSpike member of SpikeInput. The InSpike struct is defined in SpikeInput.h. Once the inputSpike has been filled with a number of input events, the user can call the SpikeInput::ScheduleQueuedSpikes() convenience function to handle the scheduling portion ReadInputBuffer(). Using this method, the example function can be simplified.

```
FooSpikeInput::ReadInputBuffer()
{
    InSpike tmpSpike;

    // get some input from a user-defined source (fooSource in
    // this example)
    while (!fooSource.Empty()) {
        tmpSpike.neuronId = fooSource.Id();
        tmpSpike.spikeTime = fooSource.Time();
        fooSource++;
        inputSpike.push_back(tmpSpike);
    }

    ScheduleQueuedSpikes();  // Schedule the spikes in spikeInput
}
```

3.2.1.2 Reading input from a file

The second method of feeding input into the network is to simply read input from a file and schedule all of the input spikes before calling Network::Run(). Two functions, ReadSpikeList(const char* fileName) and ReadSpikeDef(const char* fileName), are provided in the SpikeInput interface for this task.

ReadSpikeList(): The first line of the spike list file can be used to optionally set the units for the spike times listed in the following rows. The two choices for units are miliseconds (ms) and microseconds (us). The formatting for the switch is 'units: {unit specifier}'. The rest of the rows in the file will begin with a neuron ID followed by space-delimitted spike times in the specified units (ms is the default).

```
units: ms
1 4 10 24 30
3 3 19 29
...
```

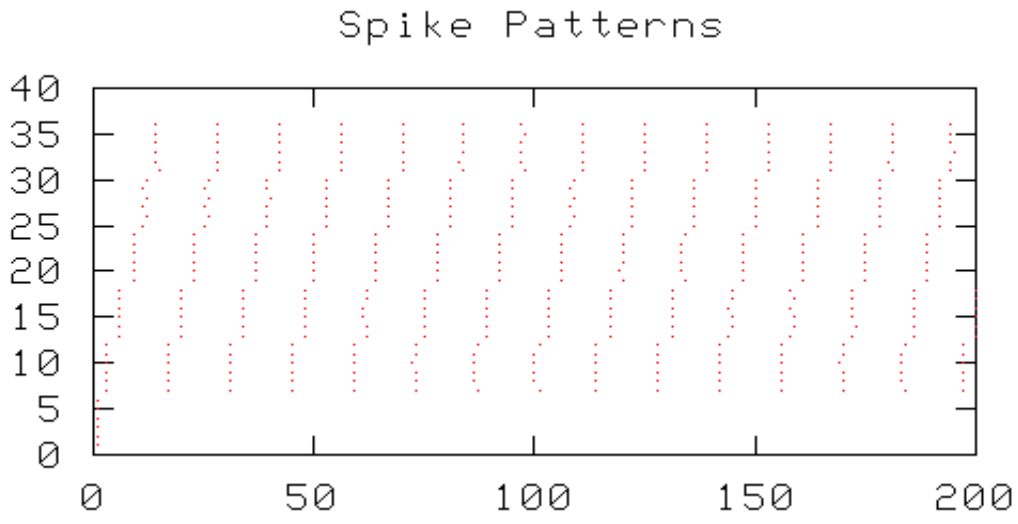
ReadSpikeDef(): ReadSpikeDef() is a virtual function, but SpikeInput provides a default implementation that can be used in many cases. The default function reads a XML formatted file that details patterns of spiking behavior for the input neurons. Currently, the only pattern available is spiking frequency, but more pattern types will be added in the future. The format for a spike frequency definition file can be found in the spikeInputFreq.dtd file.

3.2.2 SpikeOutput

The interface for SpikeOutput currently consists of a single virtual function, OutputEvent(AmIdInt neuronId, AmTimInt eventTime). This function is called every time an output neuron spikes. SimpleSpikeOutput::OutputEvent() simply writes a message to stdout. The StatisticsOutput class is also available to provide standard utilities for reporting on the activity of single neurons or groups of neurons over a period of time. Documentation on the functions available in StatisticsOutput can be found in the API documentation.

3.3 An example: SpikeLoop

The spikeloop example can be found in the samples/spikeloop directory under the main source tree. As the name implies, it is a loop of spiking neurons. Several pools (layers) of neurons are connected together in a feed-forward fashion with the output layer feeding back into the first layer. The number of neurons per layer and the number of layers can be controlled on the command line (type spikeloop –help to view a complete list of options). Spikeloop is also able to generate a gnuplot file to enable visualization of the network activity. Each dot represents a spike:



Shortly after simTime=0 we see the 6 input spikes at neuron ID1-6. After that, the spikes propagate from layer to layer until they reach the last layer from where they are sent back to the first hidden layer. This loop can continue forever.

On its surface, there is little interesting about a ring of neurons, but this type of network has a few interesting properties. The first thing to notice is that the amount of time it takes a signal to travel once around the loop is constant over multiple runs. The signal is also self-sustaining. Once a signal has been fed into the network, it will continue looping until inhibitory input stops the signal. These two properties make the spike loop a good and simple pacemaker. If the output layer of the loop feeds into another network, then this circuit can be used to feed input into the second network at regular intervals. The period of the loop can be controlled by adding or subtracting layers.

3.4 Visualizing spikes

It helps greatly if the timing of lots of spikes of lots of neurons can be visualized. To this purpose we log some or all spikes into a logfile which is then processed by Gnuplot to create an image. Gnuplot is part of many Linux distributions and can be obtained at http://www.gnuplot.org.

First, we need to turn logging on. The StatisticsOutput class provides the code for this so we need to replace the default class:

```
StatisticsOutput *spikeOut = new StatisticsOutput();
Neuron::SetSpikeOutput(spikeOut);
```

Then we turn logging on for all spikes. The log file is going to be "spikes.log" in the current directory. You can also provide a fully qualifying path.

```
spikeOut->LogSpikeTimes(string("spikes.log"));
```

To turn on logging only for selected neurons, here neuron 123, 234 and 432 we do:

```
spikeOut->AddTrace(123);
spikeOut->AddTrace(234);
spikeOut->AddTrace(432);
```

To turn the log file into a usable plot we use a input gnuplot file like this one which you might want to save to a file 'spikes.gnuplot':

```
set terminal png medium color
set output 'spikeslog.png'
set title "Spike Patterns"
show title
set size 6,1
```

```
show size
plot [0:1000] [0:70]   'spikes.log'  notitle with dots
```

---

and run:

```
bash$ gnuplot spikes.gnuplot
```

You now have an image file called spikeslog.png.

3.5 Visualizing the Topology

If your NN has properties that depend on the location of neurons in space it is very useful to visualize it. Amygdala offers a class that lets you write the whole NN or a part of it to a file that can be read by the Povray raytracer. Povary is part of many Linux distributions and can be obtained at http://www.povray.org

This code snippet writes the whole Network as it is in 'theNetwork':

```
Povray visualizer(theNetwork, 4711, string("Pacman"),
                  string("/tmp"));
visualizer.WriteNetwork();
```

This one will write a tree of all postsynaptic neurons, recursively, to a depth which is set to '2' in this example.

```
Povray visualizer(theNetwork, 4711, string("Pacman"),
                  string("/tmp"));
visualizer.WriteTree(theNetwork->getNeuron(692), 2);
visualizer.WriteTree(theNetwork->getNeuron(694), 2);
```

This code writes the axonal tree of the Neurons with IDs 692 and 694 into the file /tmp/id-4711.pov. This is a Povray input file. You can create the image by running Povray. Here is an example command line which will create a high quality image from your file:

```
bash$ x-povray +Iid-4711.pov +O4711.tga +W1280 +H1024 +V +D
+X +FT +Q9 +QR +A0.9
```

Now you have a Targa Bitmap file which you can further process with an image processor like GIMP.

4. Taking advantage of parallel Machines

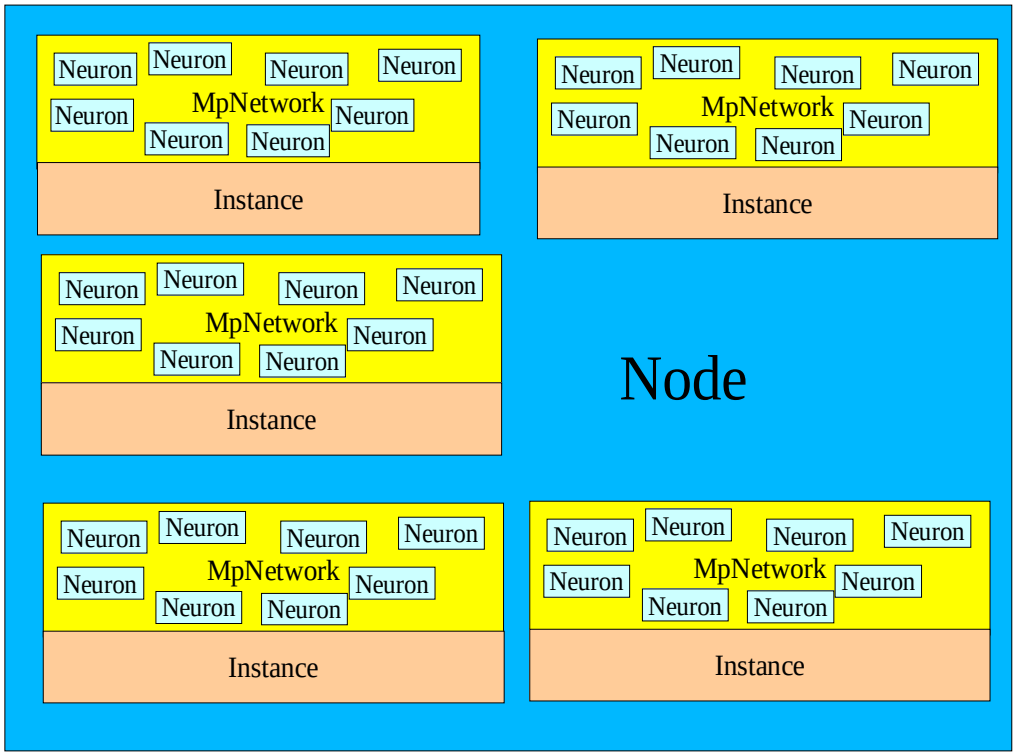
Amygdala has full support for SMP machines. The design is choosen to allow for later support of distributed memory machines and clusters. To use this, a neural

network must be split into partitions. Let's define a few terms before we get Amygdala to run on a 106-way Sun Fire:

- A **Node** is an object that contains and manages all Network objects within one process.
- A **partition** is a part of the whole NN, separated for the purpose of running it in it's own Instance
- An **Instance** object holds a partition of an Amygdala network, running one Network object in a separate thread
- A **remote spike** is a spike coming from another Amygdala instance

There is always a Node object defined when a program linked against the Amygdala library starts to run. This Node will contain a number of MpNetwork objects you define, each holding a part(ition) of the whole NN. These MpNetwork object hold their Layer objects which in turn hold their Neuron objects.

Relations of Objects



The number of MpNetworks depends on the size of the NN and on the number of CPUs in your system. To fully utilize your system you'll probably need around 2 times more partitions than you have CPUs in your system. If you have more than 10% idle time on your box you know you need more partitions, or your partitions are not symmetric enough.

Neuron IDs don't need to be unique in the whole Network, only within an Instance. But I suggest to keep them unique unless the Network is really big because this makes debugging the Network easier.

There are only a few essential functions to use SMP. They are simply listed here. The documentation of them can be found in the API docs.

```
Node::GetNodeRef()
Node::MakeNetwork()
Node::Connect()
Node::Run()
MpSpikeInput::MpSpikeInput()
Network::StreamingInput()
```

4.1 An example: SMP SpikeLoop

You find this example in the samples directory of the source tree. It is the SMP version of the spikeloop example. It runs each hidden layer and the output in a separate thread so it can utilize a number of CPUs. Don't expect a performance gain from this example. To gain performance it is necessary that the network doesn't run too much faster than real time. In other words: The network must be sufficiently big to justify SMP. Also, this is a bad design. One should try to minimize the spikes across instances. Here, all spikes are across instances. A good example of using SMP is to run several complete spikeloops that interact, each in it's own Instance.

There is a significant difference to the spikeloop example. spikeloop_smp doesn't use the NetworkLoader. The NetworkLoader class has not yet been tested with SMP. Certain convenience functions of NetworkLoader will not work over multiple MpNetwork.

After the run there will be a file 'spikes.log' which can be visualized with gnuplot by running:

```
$ gnuplot spikes.gnuplot
$ gimp smpspikeloop.png
```

5. Genetic Algorithms to evolve Amygdala Neural Networks

5.1 General Concepts

Up to now we were talking only about how to build up NNs an how to run them. But how do we create a NN that does anything useful? After all, a NN has a topology. Designing a NN with several thousand neurons where each neuron has dozens or hundreds of connections of a particular strength is an impossible task if done manually.

To facilitate the creation of large NNs we use a Genetic Algorithm (GA). Such an algorithm mimics what happens in Nature: Survival of the fittest so they can

reproduce. The new generation repeats this until the individuals are well adapted to their environment. A GA consists of a ficility to select and reproduce and a so called evaluation function. In our case, the evaluation function amounts to actually running a NN. This is an extremely computing intensive task. Because of this, we split the functionality. The selection and reproduction is done on a central server, the evaluation on clients.

We use a CGI program which is run on a HTTP server any time a client connects. This allows to serve large numbers of clients, potentially accessing the server through the Internet.

5.2 The Gene Server

As mentioned the gene server comes as a CGI program. To install it, I do on my RH7.1:

```
$ ./configure --prefix=/var/geneserver \
              --bindir=/var/www/cgi-bin
```

You need to adjust the above to your system. bindir is the directory where the server has it's cgi scripts. Your genomes will reside in subdirectories under the directory you specify with prefix

```
% make
# make install
```

Now edit the file geneserver.conf to fit your needs (described below). Then do:

```
# /var/www/cgi-bin/geneserver -g
# chown -R apache.apache /var/geneserver
```

The Web server might run under another user and group. In this case you need to adjust the above. Apache sometimes runs as nobody/nogroup.

Setting the parameters of the GA

Parameter likely_mutation_flip
Describes the likelyhood for a random bit in any gene to flip during the reproduction process. No more than one bit will flip.

Parameter population
The initial population. Reproduction takes place when a gene is requested, none is left in the outgoing directory and the number of genes in the incoming dir is larger or equal to population. Reproduction will create 'population' new genes.

The total number of 'living' genes will always be larger if the GA is run on multiple clients.

Parameter likely_crossover
The likelyhood that any female gene will be swapped with a male gene.

Parameter alltime_best
Any genome that scores higher than the value is saved into the best/ directory. This way you don't run a risk to loose the best individual which might have evolved before the algorithm converged

Parameters likely_mutation_pm and mutation_mp_amount
The likelyhood of an arithmetic mutation. This is simply the adding (50%) or subtraction (50%) of a value as set in mutation_mp_amount. Under- and overflow is possible

Parameter save_best
The best genes can go into another round of evaluation. This is useful to give an advantage to genes that deliver consistently good results.

Parameter chromosome
There can be arbitrary numbers of chromosomes with arbitraty numbers of genes in one genome. This parameter specifies the number of genes and their size in a chromosome. This has to be repeated for every chromosome. The order of the chromosome parameters is important!

5.3 Coding the client

The job of the client is to evaluate one gene at a time. This evaluation amounts to actually running the Amygdala NN. In order to make this happen, you'll have to prepare a few things. First, you have to provide an environment in which the Amygdala NN operates. This is generally the problem you are trying to solve with Amygdala. This part is entirely up to you to design and implement. The Amygdala library provides for a large part of the rest. This remaining work is the interface of your environment with Amygdala.

When you are done with coding your problem, you have to take the relevant data from your environment and feed it into the Amygdala NN. This will make the NN to start working and generate spikes. There are classes provided for you which you need to extend.

Then you have to collect the spikes and interpret them. This will usually involve changing certain parameters in the environment. To implement that, you have to derive a class from SpikeOutput and implement the method OutputEvent().

To make everything work together you have to derive your manager class from GAManager. GAManager leaves 2 purely virtual methods for you to implement:

- Setup() use this method to implement everything that's needed to get ready to run.
- GetScore() this must return the score. After the simulation finished, the score returned from this function will be reported to the gene server.

5.4 Starting up the simulation

Make sure your web server is running and the geneserver CGI program is installed

and properly configured. Now you can distribute your simulation environment to the client machines and start them up. To stop a client you simply send it a SIGHUP or a SIGINT. A SIGINT can be sent by pressing [CTRL]-C at the terminal you started the client.

If you want to run on a massivley distributed environment you have to find your own way to do this efficiently. An example is to use NFS to distribute the runtime environment and ssh to start it.

Note: Using GA with MP is not yet supported!

5.5 An example: Pacman

The Pacman environment

The Pacman code has been shamelessly stolen from a Pacman program for X, written by Fernando J. G. Pereira. This game will serve as our environment. The problem is now to find an Amygdala NN that can play Pacman.

First we have to extract some information from the environment which we can feed into Amygdala. We view Pacman as some sort of primitive live form. It has basic feelings, some good and some bad which are represented by the structure "Senses". Touching a wall is a Sense, so is smelling a ghost or smelling food. After each move, Senses is updated by the environment to reflect Pacman's situation with respect to food, walls and ghosts. This is different from how we play Pacman. We see the whole field and guide Pacman accordingly. Pacman sees only the corridor it is in.

There are 3 relevant classes in the Pacman example that are relevant to you:

- **Pacman**This class links the environment (ManIX.c) with the PacmanSpikeInput and PacmanSpikeOutput classes
- **PacmanSpikeInput** takes the current Senses from Pacman into Amygdala
- **PacmanSpikeOutput** Receives the output of Amygdala and provides it to Pacman

The file ManIX.c contains the environment - the Pacman game. It takes the movement from Pacman and returns the current Senses as a response. The inner working of these functions are not relevant to you. What is does is easily visible on the screen while Pacman runs. If you want to solve your problem with the GA, you have to provide your environment.

Feeding Pacman's feeling into the NN

To get the feeling of Pacman into the NN, we derive the class "PacmanSpikeInput from SpikeInput and implement the method ReadInputBuffer(). Please see the sources for implementation details.

Moving Pacman

The NN is supposed to find the best possible move for Pacman. To get this result we extend the class SpikeOutput and implement the method OutputEvent(). SpikeOutput collects all incoming spikes and sums up positive and negative spikes for each of the four directions.

Glueing everything together

We are now almost done. Now all components have to be arranged so they play together. This glue class is called GAManager and we need to derive our own class from it: Pacman. GAManager provides 2 purely virtual methods we must implement. The easier one is GetScore(). It is called after the simulation is over and is supposed to return the final score. We simply use the Pacman score.

Pacman::Setup() first selects the Cherrymoya gene model (the only one available now) and requests a new gene. Then the gene is processed. Then the default SpikeInput and SpikeOutput objects are replaces by the Pacman objects. Finally, the ManIX environment is initialized.

Pacman runs in discrete steps, 3 per SimSecond. The function Pacman::Step() reflects that. It first checks if the game is over, then if a level is finished. Then is is running Pacman a single step. At the end of this step the Environment updates Pacman's feelings. Those feelings are then retrieved and returned.

6. Questions and Answers

1. **What's the difference between Amygdala and other neural network simulators?**
   Amygdala models neurons more fine grained and biologically more realistic. Amygdala cptures each spike. The timing of the spikes can be consdered the protocol between the neurons on the network. Most other simulators assume that the spike rate is the protocol which is not how real neuron work.

2. **Why C++ and not [your favorite language]**
   Because C++ is fast. And accessing Amygdala from other languages is easy. From Java you can use JNI to access Amygdala. If your favorite language is plain C then simply write C using Amygdala objects and compile with g++. The statement `extern "C"` will help you linking to plain C files. Providing hooks to Java, Perl and Python is planned once the API stabilizes.

7. Copyright

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.