Development and observations of a symbolic internal state neuron.

Carl TL O’Dwyer
Bsc (Hons) Computer Science

May 2004
Development and observations of a symbolic internal state neuron.

submitted by Carl TL O’Dwyer

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinaces/#intelprop).

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:
Abstract

The Spiking Response Model is an extensive model for spiking neurons. By storing the internal states of neurons as algebraic expressions and combining it with a suitable mathematical evaluation method it should be possible to run an event based neural network. By using Interval Arithmetic to estimate neuron behaviour, through a solution finding algorithm used on the internal state, and a heap to manage the events in the system, the Spiking Response Model can effectively be implemented.
Acknowledgements

I would like to thank Dr Dan Richardson, Simon Langley, Andrew R Carrell and Florian Schanda for their comments, suggestions and help throughout this project.
Contents

1 Introduction 4

2 Physiological aspects and information 5
   2.1 The Dendrites ................................................. 5
   2.2 The Soma ................................................... 5
   2.3 The Ionic Nature of Neurons .................................. 6
   2.4 The Axon .................................................... 6
   2.5 Synapses ..................................................... 6
   2.6 Spike trains ............................................... 7

3 Mathematical Tools 8
   3.1 Interval Arithmetic .......................................... 8
   3.2 The Heaviside Function ...................................... 9
   3.3 Approximating solutions to equations using Interval Arithmetic 9

4 Mathematical Models 12
   4.1 First generation neural networks .............................. 12
   4.2 Second generation neural networks ........................... 12
   4.3 A simple spiking neuron ...................................... 13
   4.4 The Spiking Response Model .................................. 13
      4.4.1 The Action Potential and after-potential ............... 14
      4.4.2 Summation of Synaptic input ............................ 14
      4.4.3 The external driving current ............................ 14
      4.4.4 Generality of the Model ................................. 14
      4.4.5 Algebraic Internal State ................................. 15

5 Implementation 17
   5.1 Resources .................................................. 17
      5.1.1 Unix like platform ...................................... 17
      5.1.2 GiNaC .................................................. 17
   5.2 Implementing Interval Arithmetic ............................ 17
   5.3 Events Management .......................................... 18
      5.3.1 Why Event Driven? ....................................... 18
      5.3.2 The Heap ................................................ 18
Chapter 1

Introduction

Neural nets are, fundamentally, an attempt to copy the computational system that has developed and evolved in living creatures. Biological neural nets are fast, error tolerant and adaptable. These qualities are what make neural nets attractive. Early models used a very simplified version of a neuron (the McCulloch-Pitts model) due to hardware and information limits at the time. Later models, based on different principals, extended the capacities of artificial nets greatly (sigmoidal nets), but with more recent data, it has become apparent that some biological neurons function in a different manner to the rate based encoding of sigmoid-gate neurons. Vision, for example, responds very quickly to stimulation, around the order of a 100ms. Such a short period of time is insufficient for a rate based encoding system to calculate the large quantity of information that the biological counterparts can handle. As such, a different approach was explored, where the net functions on a pulse based encoding (actually similar to the McCulloch-Pitts model in some ways). Such neural nets are made up of what is called “spiking neurons”, where importance is placed not on the number of spikes generated by a neuron, but the timing of these spikes [4].

The Spiking Response Model is a mathematical model for representing such neurons. With it, it should be possible to have a symbolic internal state for each neuron. If this is the case, how would the simulation be performed? What mathematics would be needed?
Chapter 2

Physiological aspects and information

A neuron is an individual cell; one building block of a nervous system. Across all species the fundamentals of their behavior differ only slightly. Generally speaking, there are three parts to a neuron, the axon, the soma and the dendrites.

2.1 The Dendrites

The idea behind a nervous system is that simple signals are sent from one neuron to the next, and given the right conditions, another signal is then generated at this receiving neuron and sent on. Of course, in reality, there are a huge amount of neurons all interlinked (roughly $10^4$ neurons per cubic millimeter in the cortical region). The part of the neuron that receives these signals is the dendritic tree. The dendrites generally fan out, and are connected to the axons of other neurons through links called synapses. The received signals are called either EPSP (Excitatory Post-Synaptic Pulse) or IPSP (Inhibitory Post-Synaptic Pulse), or just PSP when not distinguishing. This difference is dependent on the type of synapse, and not the neuron itself, and is characterized by the influence the pulse has on the soma. The dendrites transport these pulses to soma, but they can suffer some modifications when the distance traveled along the dendrite increases.

2.2 The Soma

The soma performs the critical task of integrating all the various pulses received. When the state of the soma reaches a threshold, the soma produces an Action Potential, that is carried along the axon. After an action potential is generated, the soma remains insensitive to any new pulses received for a certain period, known as the absolute refractory period. Once passed this, the
neuron regains its sensitivity over a certain amount of time, this is known as the relative refractory period. These events are due to the creation of the action potential, which polarizes the neuron.

2.3 The Ionic Nature of Neurons

In order to grasp some of the phenomena happening inside a neuron, it is important to have a basic understanding of the way in which a neuron works. When a neuron is at resting potential (any effects from previously received pulses have dissipated, and the neuron hasn’t fired recently either), its internal voltage is negative. When a PSP is received, charged ions enter the neuron through the dendrites and travel to the soma. When enough positive ions are present in the soma, the internal voltage will increase until it hits the threshold. This causes a large reaction depolarising the neuron, and also creating an action potential. A certain amount of time is required to allow the balance of ions to return to normal values. It is also worth explaining the difference between EPSP and IPSP here; as mentioned earlier, PSP is a flux of ions entering the neuron, in the case of an EPSP these ions are positively charged, while for an IPSP these ions are negatively charged, thus lowering the internal voltage of the neuron.

2.4 The Axon

Once an Action potential is generated by the soma, it travels along the axon, and arrives at other neurons. The Axon has many branches, and at the end of each branch is a synapse. Generally speaking, branches don’t end far from the neuron, but there are sometimes exceptions. Also, one neuron can be linked to another through many branches. The synapse at the end of the branch is what allows for the link between two neurons. It is also worth noting that an increased distance between two neurons means an increased delay between the spiking of pre-synaptic neuron and the corresponding effects on the post-synaptic neuron.

2.5 Synapses

The synapse is another key element of a neural network. Most synapses are of a chemical nature, and store what is known as a neurotransmitter in pockets called vesicles. When an AP arrives, it forces some of these vesicles to be opened, releasing the neurotransmitter, which in turn acts on the post-synaptic neuron and delivers the PSP. The magnitude of the PSP defines what is called the “synaptic potential”. It is known that the synapses are responsible for a large part of the learning factors in the nervous system, but how exactly this is achieved is still not entirely clear.
2.6 Spike trains

A spike train is simply a sequence of APs generated by a neuron. Each AP normally has a duration of 1-2 ms, and the shape of the AP doesn’t vary, so its form doesn’t carry any information. This is crucial, as it means that the information is contained in the sequence of the spikes. Under normal stimulation the APs are quite separated, and even with a much stronger input, there should always be a gap between the APs, because a neuron’s absolute refractory period is longer than the duration of an AP.

Description based on [8].
Chapter 3
Mathematical Tools

3.1 Interval Arithmetic

Interval arithmetic is the adaptation of standard arithmetic for working on intervals instead of single numbers. [2]

\[ 5 + 3 = 8 \]
\[ [1, 2] + [1, 5] = [2, 7] \]

By replacing a number with an interval, it is possible to bound the domain of a function, although interval arithmetic only guarantees to find an interval which includes the function's domain.

For a better understanding of what is done, here are the four definitions of the basic operators, where a, b, c, and d are real numbers.

\[ [a, b] + [c, d] = [a + c, b + d] \]
\[ [a, b] - [c, d] = [a - d, b - c] \]
\[ [a, b] \times [c, d] = [\min\{a \times c, a \times d, b \times c, b \times d\}, \max\{a \times c, a \times d, b \times c, b \times d\}] \]
\[ 1/[a, b] = [1/b, 1/a] \text{ if } a > 0 \text{ or } b < 0 \]
\[ [a, b]/[c, d] = [a, b] \times 1/[c, d] \]

Another very important operator is the exponential function. Luckily, it is monotone over \([+\infty, -\infty]\), it is very easy to determine its equivalent in interval arithmetic:

\[ e^{[a, b]} = [e^a, e^b] \]

Also, the operator “Absolute Value” is needed, but follows some different rules:

\[ |[a, b]| = [|a|, |b|] \text{ if } a > 0 \text{ or } b < 0 \]
$|a, b| = [0, max\{|a|, |b|\}]$ if $a <= 0$ and $b >= 0$

Now that the basic tools have been introduced, their weakness should also be examined; boundary overestimation is a real problem with interval arithmetic.

Although we know that $h([-1, 1]) = [0, 0]$, it will still be calculated as $[-2, 2]$. As the size of the interval decreases, the precision of the bounding increases.

Care must be taken when using interval arithmetic, especially when defining more complex functions as range calculated can sometimes be much larger than the actual range.

3.2 The Heaviside Function

The Heaviside function generally plays the role of a switch in mathematics. It allows for functions to be turned on only after a certain point in time. There is one point of debate concerning the Heaviside function, and that is its value at 0. Some assign it the value 0, others the value 1 and also sometimes the value 0.5. In actual fact, this isn’t a problem, as the function will only be used with interval arithmetic, and therefore the interval $[0,0]$ is probably best mapped to the interval $[0,1]$.

It is important to point out that it is an assumption that the Heaviside function can be adapted to interval arithmetic, although because of the conservative nature of interval arithmetic, this is not a problem, as the Heaviside function is guaranteed a range of $[0,1]$.

3.3 Approximating solutions to equations using Interval Arithmetic

Interval arithmetic can be used to approximate the solution of an equation over a given interval. An estimate of the range of the function is calculated for that interval, and if the range includes the desired value, then it is split into two smaller intervals, and each of these is checked. When the estimated
range of an interval doesn’t contain the desired result, the that interval, and all intervals contained within it can be ignored, as the can not possibly contain a solution to the equation. As with other approximation methods, the more the fundamental steps are repeated, the more accurate the solution found becomes.

The algorithm specified below is recursive, and uses sub-division of the original interval.

\( u(t) \) is an internal state as a function of \( t \).
\( \delta \) is the precision indice.
\( T \) is the threshold.
The input is an interval \([a,b]\).
The output is an interval.

```
search([a, b])
{
    u([a, b]) = [c, d];
    if(d > T)
        if(b - a < \delta)
            return [a, b];
        else
            search([\frac{a+b}{2}, b]) = [c, d];
            if([c, d])
                return [c, d];
            search([a, \frac{a+b}{2}]) = [c, d];
            if([c, d])
                return [c, d];
        return false;
}
```

An example to demonstrate the process now follows: Given the equation \( e^x - 1 = 2 \)
gives us a range of $[0, 6.4]$. A search is then performed on $[0, 1]$, but returns false so a search on $[1, 2]$ is performed, then a search on $[1, 1.5]$ and finally a search on $[1, 1.25]$. This last search returns the range $[1.7, 2.5]$ and as $1.25 - 1 < 0.5$ the interval $[1, 1.25]$ is returned as the solution.

With a smaller precision indices, it would of course take more steps to solve, and it is possible that some branches only yield a false answer after a few levels of recursion, but the size of the intervals decrease on a logarithmic scale, which can lead to some very quick solutions.

The algorithm has some properties worth taking note of. The first is that the algorithm will always terminate; either because the interval checked contains no solution, or because the intervals can not get too small, as they are limited by the value of $\delta$. The second property is that a negative answer, ie no firing time found, is always valid, while a positive answer can be invalid. This is because of the conservative nature of interval arithmetic, and will only happen in cases where the curve comes very close to the threshold.
Chapter 4

Mathematical Models

4.1 First generation neural networks

The first generation of neural networks consisted of the McCulloch-Pitts model with only a binary output, these have been used in large scale networks like multi-layer perceptrons. The neuron works by summing its weighted input, and if this is above the threshold value, the neuron outputs a binary ‘high’ signal. This model works in what is known as discrete time. The simulation progresses by steps, and in each step every neuron is processed.

The greatest achievement of this kind of model is that it is possible to compute any function with boolean output. This means the model is universal for digital computations. Another characteristic of such networks is that they remain synchronised with very little overhead.[6]

4.2 Second generation neural networks

The second generation of neural networks differs radically from it’s predecessor. The output from a neuron is not dependent on a step or threshold function, and it is no longer a binary output. This makes the second generation suitable for analog input as well as analog output. Instead of a threshold function, the internal state of the neuron is computed using a sigmoid or hyperbolic tangent function. These neurons also work in a discrete manner.

Neural Networks of this generation are also much more powerful than the first generation; when coupled with threshold based neurons as the output layer, the network is universal for digital computation using less neurons than an equivalent network of the first generation. A second generation network can also approximate any analog function with arbitrary precision, so such networks are also universal for analog computations.[6]
### 4.3 A simple spiking neuron

The Leaky integrate and fire is one of the most widely used spiking neurons, as well as one of the simplest [7]. Because of neurons electrical nature, the lif is modeled by a circuit, which in its most basic form, contains a capacitor and a resistor.

The current $I(t)$ is split across the resistance and capacitor, giving the equation $I(t) = I_R + I_{cap}$. $I_R$ is determined using Ohms law, and gives us $I_R = u/R$ where $u$ is the voltage across the resistor. $I_{cap}$ is the current that charges the capacitor and is given as $I_{cap} = Cdu/dt$ with $C = q/u$. $q$ is the charge of the capacitor and $u$ the voltage across it. So we now have

$$I(t) = u(t)/R + Cdu/dt$$

with $\tau_m = RC$, which is the time constant of the leaky integrator.

This formula describes the behavior of the neuron between firing times, which also have a set of rules. In the lif, APs are not described explicitly, and are only characterized by their firing time. When a lif neuron fires, its internal state is set to some value $u_r$. This is the basic lif model, but in a slightly broader version, and absolute refractory period is also included by stalling the integration for a time $\delta_{abs}$. But how exactly does stimulation from other neurons come into this framework?

$I(t)$, the input current, is actually the sum of all stimulation from the presynaptic neurons, so for a neuron $i$ the formula is:

$$I_i(t) = \sum_j W_{ij} \sum_f \alpha(t - t^{(f)}_j)$$

$W_{ij}$ is the efficacy of the synapse from $i$ to $j$.

$\alpha(t - t^{(f)}_j)$ is the time course of the input current. It can be expanded to include many extra features, to enhance the simulation.

### 4.4 The Spiking Response Model

The SRM is a mathematical model to represent spiking neurons. It is composed of three main elements. It is represented by the following formula, taken from [8]. $u_i(t)$ is the internal state of neuron $i$ at time $t$.

$$u_i(t) = \eta(t - \tilde{t}_i) + \sum_j w_{ij} \sum_f \epsilon(t - \tilde{t}_i, t - t^{(f)}_j) + \int_0^\infty \kappa(t - \tilde{t}_i, s) I^{ext}(t - s) \, ds$$

(4.5)
4.4.1 The Action Potential and after-potential

The function $\eta(t - \hat{t}_i)$ represents the form of an action potential in the neuron internal state, and the neurons behavior after it has generated an action potential. $t$ is the current time, and $\hat{t}_i$ is the time of the last firing. According to [8] in a simplified model, only the exact time of firing is important, as the shape of the action potential is always the same, and thus carries no useful information. Therefore, $\eta$ represents the neurons behavior after a spike is generated, it is called the “reset” function. The extra advantage of this is that it is no longer necessary to have a dynamic threshold, as $\eta$ ensures that it is virtually impossible to cross the threshold before the absolute refractory period is over. This is also called the reset kernel.

4.4.2 Summation of Synaptic input

The function $\epsilon_{ij}(t - \hat{t}_i, t - t_j^{(f)})$ represents the shape of the input from neuron $j$ to neuron $i$. The first argument allows for the relative refractory period phenomena that has been observed in natural neurons. While the second is the set of spikes from neuron $j$, where $f$ is the indices for the spikes. The term $W_{ij}$ is the synaptic efficacy from neuron $j$ to neuron $i$, or better known as the weight of neuron $j$ for neuron $i$. This part of the equation represents the effect of the pre-synaptic spikes received on the internal state of the neuron.

4.4.3 The external driving current

The integral in the SRM function represents how an external input affects the system. This is needed in cases where the neurons can be stimulated directly, and not just by input from pre-synaptic neurons. The function $K(t - \hat{t}_i, s)$ represents the shape of this input, while $I_{ext}(t - s)$ represents the value of the external current used. The argument of $(t - \hat{t}_i)$ for $K$ is again to allow for the relative refractory period of the neuron. The value $s$ is in fact the time since the stimulus was applied, so if a pulse was applied to the neuron at time $t'$ then $s$ is calculated with $(t - t')$, and $I_{ext}(t - s)$ is in fact just $I_{ext}(t')$.

4.4.4 Generality of the Model

Because the SRM is such a broad model, it is capable of performing as a lif network. The lif is driven by an external current $I_{ext}$ and post synaptic current pulses $\alpha(t - t_j^{(f)})$, the potential $u_i$ is given by

$$
\tau_m \frac{du_i}{dt} = -u_i(t) + R \sum_j W_{ij} \sum_f \alpha(t - t_j^{(f)}) + RI_{ext}^i(t) \tag{4.6}
$$
but the $RI_{ext}^i(t)$ can be dropped as we are only interested in the case where the network receives post synaptic current. So we are left with

$$\tau_m \frac{du_i}{dt} = -u_i(t) + R \sum_j W_{ij} \sum_f \alpha(t - t_{j}^{(f)})$$

We need this to be a function of $\tau$, and so the above equation is integrated with the condition $u(t_i) = u_r$, which allows for the reset on spiking behavior of a lif neuron.

$$u(t) = u_r \exp\left(-\frac{t - t_i}{\tau_m}\right) + \sum_j W_{ij} \sum_f \frac{1}{C} \int_0^{t - t_i} \exp\left(-\frac{s}{\tau_m}\right) \alpha(t - t_{j}^{(f)} - s) \, ds$$

$$= \eta(t - t_i) + \sum_j W_{ij} \sum_f \epsilon(t - t_i, t - t_{j}^{(f)})$$

$$= \eta(t - t_i) + \sum_j W_{ij} \sum_f \epsilon(t - t_i, t - t_{j}^{(f)})$$

so we have

$$\eta(s) = u_r \exp\left(-\frac{s}{\tau_m}\right)$$

$$\epsilon(s, t) = \frac{1}{C} \int_0^s \exp\left(-\frac{t'}{\tau_m}\right) \alpha(t - t') \, dt'$$

we now have formulae for the reset kernel, and the PSP kernel, but the PSP kernel still needs to be made explicit. So using a convenient definition of $\alpha(s)$ we get:

$$\alpha(s) = \frac{q}{t_s} \exp\left(-\frac{s}{t_s}\right) \Theta(s)$$

with $q = C = 1$, the integration in $\epsilon$ yields:

$$\epsilon(s, t) = \frac{\exp\left(-\max(t-s,0)\right)}{1 - \frac{t}{\tau_m}} \times [\exp\left(-\frac{\min(s, t)}{\tau_m}\right) - \exp\left(-\frac{\min(s, t)}{t_s}\right)] \Theta(s) \Theta(t)$$

(4.13)

So there are now explicit formulae for both the reset kernel and the PSP kernel, meaning that the lif model is covered by the SRM. (for full compliance, the external driving current $I_{ext}$ should also be transformed into an explicit form.)

4.4.5 Algebraic Internal State

I wish to implement the SRM, by representing the internal state as a complex polynomial in $t$ and $t_i$. Therefore I need to find adequate polynomial representations of $\epsilon$ and $\eta$. Because $\kappa$ only concerns outside stimulus, I believe that I can ignore it at least for the time being. The difficulty in finding such representations is they must be correct for all values of $t$, not just for $t > t_i$. An example of this would be using the function $-e^{(t-t_i)}$ for $\eta$, but if $t$ were a lot smaller than $t_i$, $\eta$ would generate a very large negative number.
Although the case where $t < \hat{t}_i$ is very unlikely to happen, the same problem can occur in $\epsilon$ if $t'^{(f)}_j$ is greater than $t$, which is very likely to happen.

So to be able to use the Spiking Response model with a complex polynomial in $t$ and $\hat{t}$ as internal state, I must find polynomials in $t$ and $\hat{t}$ that tend, for $\epsilon$, towards 0 at $+\infty$ and $-\infty$, and for $\eta$ towards 0 at $+\infty$. 
Chapter 5

Implementation

5.1 Resources

5.1.1 Unix like platform

The program was intended to run in a Unix like environment as this is what I am the most familiar with. The various tools I wished to use, such as emacs, were all available, and the Computer Algebra Library I wished to use was also intended for use in such an environment. Because C++ was the language of choice for the project, the availability of related tools was most certainly a key point.

5.1.2 GiNaC

Due to the mathematical nature of the objectives, it was obvious that some sort of algebra package would be required. As it was both difficult and time consuming to write one from scratch, I instead used one already available that could be adapted for my purposes. Of course, there were other requirements.

The system used had to be easy to integrate into a C++ program, and also potentially be fast. In both these aspects, GiNaC proved to be a very good choice. As it is not a full Computer Algebra System, it was much easier to use inside of the project source code, but still allow for a large amount of flexibility.

GiNaC allows for very easy manipulation of algebraic formulae and has the extensibility needed to add new number types like intervals.

5.2 Implementing Interval Arithmetic

One of the main objectives of the project was to determine whether Interval Arithmetic could be used to properly estimate the firing time of a neuron
when the internal state is represented by a polynomial containing many exponentials, which can be quite difficult to solve exactly.

GiNaC was unfortunately missing a few essential tools, the first set being the basics of interval arithmetic. Fortunately source code was made available by Simon Langley that allowed GiNaC to perform these basics. The code did not, however, allow for the use of exponentials or the absolute value operator. Another important tool was the heaviside step function.

Fortunately, GiNaC allows for new functions to be created quite easily. Implementing exponentials and the absolute value operator was quite trivial, only requiring a method for intervals to be defined. Implementing the Heaviside step function was a little more complicated, as no prior version was present, but it presented no real problem. GiNaC provides the necessary procedures to declare new functions, and be able to use them inside expressions.

### 5.3 Events Management

#### 5.3.1 Why Event Driven?

To increase the speed of such networks one can increase the time step between each calculated state. This unfortunately reduces the accuracy and computational power of the net, due to less precision in the timing of events. A second, cleaner solution, is to turn to an event driven simulation, instead of a state driven simulation. This means that only the neurons that receive input are processed. In a very large network, where only a 10th of the neurons are firing at any given time, this is a very large speed up [5]. It does, however, mean that the neurons themselves become larger amounts of computation. This is because a lot of calculation will go into predicting when events happen. In my system, an event would represent a Neuron firing.

#### 5.3.2 The Heap

One of the main issues with such a system is that predictions often change, creating two different problems. The first is that predicted events that have been voided must be dealt with. The second is arranging the events so that they will be processed chronologically. When only dealing with a small amount of events, this isn’t really a problem, but to run larger scale simulations requires a system that can scale well. There are of course a lot of different options; [5] describes a queue based system, while [1] introduces the idea of using multiple pools, each corresponding to a particular interval of time. A modified quick sort algorithm is then used to get the earliest element out of the leading pool. The amount of computation is reduced if the size of the pools is kept small enough for the cost of the search to be
cheaper than that of an ordered queue. I wish to take another path, and use a heap data structure to store the events.

The key used to order the events is the predicted firing time associated with that event, with the earliest event at the top of the heap. Adding an event consists of placing the event at the bottom of the heap, and pushing it up the heap as long as the parent nodes have a later firing time. There is no need for a heapify function, as the data is ensured to be in a heap because elements are added one by one to an empty heap.

Now comes the question of how to remove an event from the heap, yet keep the heap intact. If instead of removing an event, it is replaced by the last event in the heap, which would be one of the later events in time, and then this event is pushed down the heap to a valid place, the integrity of the heap is maintained. Of course, this doesn’t quite solve the problem of obsoleted events.

If each neuron has a marker that is changed whenever a new event is created for that neuron, and each event stores a copy of the marker from when it was created, then determining a valid event comes back to comparing the neurons current marker and the one stored in the event. If they match, then the event is valid, otherwise the event is obsolete, and should be discarded. The simplest way of using this is to only perform the comparison on an event that is about to be run; if it is invalid it is simply dropped, otherwise the associated neuron is fired. Although the cost might seem high, having to store and deal with all of the obsolete events until they are removed one by one, it is actually quite low. If, for example, there are 15 times more obsolete events than valid ones, this only represents 4 extra basic operations more than if the heap did not have those obsolete events in it. This is because both adding and removing events from the heap has a complexity estimate of $O(n) = \log(n)$. [3]

The actual load placed in the heap is dependent on the simulation; the amount of input spikes, the size of the network, and the average amount of connections all affect the amount of items in the heap. Sharp increases and decreases in the load is very usual. For example, if a highly connected neuron fires, it can potentially create an event for each of the neurons it is connected to thus leading to a very brusque increase in the amount of events in the heap. This also explains the rapid decreases which are due to obsoleted events left in the heap being discarded; when two or three highly connected neurons fire one after the other, then multiple events might be created for a same neuron, of which only one is valid and the others must be discarded. Because the majority of neurons are excitatory, then the valid event tends to have an earlier firing time, and the obsoleted events end up grouped together chronologically.
5.3.3 Running an Event

Now that only valid events are guaranteed to be run, it is worth looking at what exactly happens when an event is run. As mentioned previously, an event represents the firing of a neuron, so the first task is to apply the reset kernel to the neuron. Then the next firing time of this neuron has to be estimated. After this, each of the synaptic connections has to be taken care of. This constitutes the steps needed when an event is run.

5.4 Sending PSPs

Each neuron contains a set of connections, where these connections represent the synaptic links between neurons. When a connection is told to deliver a PSP, it takes the polynomial representation of a PSP and substitutes the variable $t$ with $t - \text{firingtime} - \text{delay}$. This new polynomial is then multiplied by the weight factor of the connection, and then added to the receiving neurons internal state, whereupon a new estimate is calculated for the receiving neuron. Both the delay and the weight are values stored in the connection.
5.5 Estimating Firing Times

The key feature of an event based simulation is estimating the next firing time of a neuron. With a polynomial internal state, it would be nice to just solve the equation, but unfortunately that is a rather complex issue, especially when the internal state contains many different PSPs, and also there can be multiple solutions to the problem.

5.5.1 The conditions for a valid solution

When searching for an estimation of the next firing time, care must be taken to respect some rules that facilitate the process. The first is that there is no point in searching for an estimation which is earlier than the last firing time. Also, if a valid estimation exists that is earlier than the beginning of the search time region, it is not necessary to perform the search. This last point is useful if, for example, a PSP arrives at the neuron later than the next predicted firing time, as it means there is no work to be performed. Of course, it the last estimation is inside the time interval searched, then the new estimation should be used. If the previous estimation is inside the search interval, yet no new estimation is found, it is then necessary to search after the time interval; this is the case where an IPSP cancels out some EPSPs. The time interval used for the search can be limited to the interval where the arriving PSP isn’t null for the case of a new PSP, or in the case of neuron that fired, the interval should be large enough to cover any possible PSPs that could have arrived at the neuron.

5.5.2 The largest possible interval to search

Because the system is event driven, time moves forward in irregular steps. The current simulation time is actually determined by the firing time of the current event, they are the same thing. Therefore working out the latest possible PSP to arrive at a neuron is quite easy, because the maximum delay associated with the links between neurons is known. If neuron A fires at time T, and the link from A to B has a delay value of D, then the PSP arrives in B at time T+D. If D is the greatest delay value for a link arriving at B, then the PSPs from all other neurons fired at time T that link to B will arrive before T+D. So if D is the greatest delay value for the entire network, then all PSPs must arrive at or before time T+D, independently of whichever neurons A and B happen to be. This is very useful because it means there is a good approximation of where to end an interval when searching for the next firing time; if there are no PSPs past time T+D+L, where L is an offset to compensate for the length of a PSP, the neuron can’t possibly fire after time T+D+L.
5.5.3 Using Interval Arithmetic

As mentioned earlier, solving equations involving lots of exponentials is not very easy, and although there exist various methods to approximate a solution, Interval Arithmetic has some extra advantages. The first is that interval arithmetic can determine very quickly if a solution even exists; when an interval is tested, if the higher end of the resulting interval is lower than the threshold, then there is no possible solution inside that interval. The second advantage is that interval arithmetic is easy to perform.

But there is one major disadvantage, and that is the fact that interval arithmetic doesn’t return the exact range of a function, but instead an interval that includes that range, so the roof of the returned interval is sometimes far too high. This means that firing times can seem to exist in a particular interval, when in fact there are none. The amount of error does reduce as the size of the tested interval is reduced, but only when the intervals start to get very small.

5.5.4 Clipping the Internal State

To avoid expression swell, and at the same time increase performance, elements in the interval state that no longer influence it enough to be useful are dropped. To determine whether an element must be dropped, its range is calculated. The maximum and minimum are then compared to boundaries, and if both are close enough to 0, then the element is removed.

This “clipping” of the internal state is very useful when the simulation has been running for a while, as it removes the older PSPs that are no longer relevant.

5.5.5 Analysis of the Algorithm

The original algorithm was recursive in nature. Although this is not a problem, I preferred to implement an iterative version, as it is easier to maintain.

\[ u(t) \] is the internal state as a function of \( t \).
\[ \delta \] is the precision indices.
\( T \) is the threshold.
The input is an interval \([a, b]\).
The output is an interval.

\[
\text{Ans} = \text{NULL}; \\
push [a, b] \text{ onto interval stack}; \\
\text{while (interval stack not empty)} \\
\{ \\
\quad [a, b] = \text{pop top of stack}; \\
\}
\]
\[ [c, d] = u([a, b]); \]
\[ \text{if}(d > T) \]
\[ \text{if}(d - c < \delta) \]
\[ \text{Ans} = [a, b]; \]
\[ \text{break}; \]
\[ \text{else} \]
\[ \text{push} \left( \frac{a+b}{2}, b \right) \text{ onto stack;} \]
\[ \text{push} \left( a, \frac{a+b}{2} \right) \text{ onto stack;} \]
\]

The use of a stack is important. It ensures that it is the sub intervals of the previous interval that are checked. This is effectively equivalent to the recursive version.

The advantage of this algorithm is that it is quite simple. Its design is based on the idea that the interval arithmetic is reasonably accurate, and when this is the case, there should be very little difference between finding an estimate near the beginning of the tested interval and one near the end. Because it divides the the test interval by 2 each time, it is capable of finding an estimate in \( \log(n/p) \) steps, where \( n \) is the length of the first interval tested and \( p \) is the level of precision desired. Generally this is not the case, as some intervals need to be looked at very closely to determine that they do not contain any possible firing times. This is due to the nature of interval arithmetic, for example if two PSPs were in an interval, but not overlapping, the range of that interval would still include the sum of the maximums of each PSP. Such a value would be greater than the maximum of either PSP, and could also potentially be above the threshold, while the internal state never actually crossed it. On the other hand, when an interval is empty, or sparsely populated by PSPs, it only requires very few steps to determine that there are no firing times, and in this way, large portions of the original interval can be disregarded quite quickly.

Of course, when the interval arithmetic fails to be reasonably accurate, then the algorithm must perform more basic steps to determine the presence of a possible firing time.

### 5.5.6 Demonstration of the algorithm

In order to demonstrate the usefulness of this algorithm an equation that closely resembles the internal state of a neuron has been chosen. The first aspect of the algorithm to be looked at is the size of the search interval, and how the algorithm breaks it down. The following table shows the intervals, and their calculated range. Although the precision indice is set to 0.01, for this simulation it was enforced on both the width of the interval and the width of the corresponding range. If this had not been the case, the algorithm would have taken 18 steps, and not 22. The earliest interval is
always at the top of the stack. This ensures that when a solution is found, it must be the earliest solution possible. Also, the further down the stack, the larger the intervals get. If a solution is found at the top of the stack, then these larger intervals need not be checked.

In the first implementation of this algorithm, each interval was made of two entries in the stack, which added a lot of bloat to the stack because there were many double entries of the same value, which also increased the operations to be performed at each step. The second implementation only stores the values once, which resulted in less memory usage, and a faster execution of the algorithm. This was possible because the the intervals not yet checked make up a continuous interval, so there is no need to worry about discontinuities.

The second important aspect of the algorithm is how fast it converges on a plausible solution. There are two important and distinct values of precision when determining a solution. What is probably the most important one is the width of the desired solution interval, while the other is the range corresponding to this solution. Figure 5.3 shows how the width of the interval decreases in an exponential fashion, while the width of the range does not present any particular pattern. The algorithm finds the interval solution \([6.34, 6.34]\). A Computer Algebra System, in this case Maple, finds the solution to be 6.34. The algorithm has effectively returned the correct answer, to two decimal places in precision.
<table>
<thead>
<tr>
<th>Interval</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>38.58</td>
</tr>
<tr>
<td>6</td>
<td>22.29</td>
</tr>
<tr>
<td>6</td>
<td>14.15</td>
</tr>
<tr>
<td>6</td>
<td>10.07</td>
</tr>
<tr>
<td>6</td>
<td>8.04</td>
</tr>
<tr>
<td>6</td>
<td>7.02</td>
</tr>
<tr>
<td>6</td>
<td>6.51</td>
</tr>
<tr>
<td>6</td>
<td>6.25</td>
</tr>
<tr>
<td>6.25</td>
<td>6.51</td>
</tr>
<tr>
<td>6.25</td>
<td>6.38</td>
</tr>
<tr>
<td>6.25</td>
<td>6.32</td>
</tr>
<tr>
<td>6.32</td>
<td>6.38</td>
</tr>
<tr>
<td>6.32</td>
<td>6.35</td>
</tr>
<tr>
<td>6.32</td>
<td>6.33</td>
</tr>
<tr>
<td>6.33</td>
<td>6.35</td>
</tr>
<tr>
<td>6.33</td>
<td>6.34</td>
</tr>
<tr>
<td>6.33</td>
<td>6.34</td>
</tr>
<tr>
<td>6.34</td>
<td>6.34</td>
</tr>
<tr>
<td>6.34</td>
<td>6.34</td>
</tr>
<tr>
<td>6.34</td>
<td>6.34</td>
</tr>
</tbody>
</table>

Figure 5.3: Interval and Range boundaries
Figure 5.4: Width of the Range and Interval
5.6 Complexity Analysis of the System

I will now attempt a rough estimate of the complexity of my system.

5.6.1 One Estimation

When calculating one estimation, if a basic step is considered to be checking whether the calculated interval contains the threshold, then the complexity ranges from 1 in the best case right up to $2 \times (l/p)$ where $l$ is the length of the interval and $p$ the desired precision. It isn’t feasible to give an average, as this complexity highly depends on both the internal state being checked, and the interval it is being checked on.

5.6.2 One Event

One Event is the firing of a neuron, so the amount of estimations performed is $1 + c$ where $c$ is the number of connections. Although it is impossible to give an exact value to $c$, one would expect to have at most $c = n$ where $n$ is the amount of neurons in the network.

5.6.3 Dependence on Network Parameters

Now that the complexity of one event is roughly known, the next step is to determine how many events there are. Unfortunately this is even more difficult than giving an average for the complexity of an estimation. If instead, each neuron is considered to fire as much as it can, then the amount of events becomes $n \times t/r$ where $t$ is the total time interval of the simulation, and $r$ is the absolute refractory period of the neurons. This estimate is, in most cases, far too great, as some neurons will only fire once or twice in the entire simulation.

5.6.4 General Estimate

Now that there are some very rough estimates for different parts of the system, it is possible to give a general one:

$$(1 + n) \times 2 \times (l/p) \times n \times t/r = 2 \times (l/p) \times n \times t/r + 2 \times (l/p) \times n^2 \times t/r \quad (5.1)$$

It should be noted that this is only a very rough estimation, and that anything with a better approximation will already start becoming specific to a particular network configuration.

5.7 Observations from a sample network

What kind of properties and behavior are achieved using the developed system? There is, of course, a fundamental dependency between the behavior,
the input and the configuration and parameters of the network used. The first example is composed of 5×5×5 core neurons, with 1 input neuron connected to 10% of the core neurons, and 1 output neuron connected to all of the core neurons. The network has been setup in a way that is very sensitive, this allows for some of the basic behavioral patterns to be observed using very simple input spike trains.

5.7.1 Reaction to one Spike

Figure 5.4 shows the two spikes of each of the 4 simulations. The first spike is the input spike, while the second is the output spike. The gap between these two spikes is the same for each simulation. This shows that the neural network is reacting to the input spike, but not its time of occurrence. This is of course the desired behavior for such input.

5.7.2 Reaction to two Spikes

Using a single spike isn’t very informative of the behavior of a neural network, after all, one of the main idea’s concerning spiking neural nets is that information is stored in the relative timing of the spikes. It should then be possible to see different output patterns using only 2 spikes in the input spike train.

Figure 5.5 shows the input firing times, while figure 5.6 shows the firing times of the output neuron. The pattern in relative timing amongst the spikes of a spike train are different in cases 1, 2 and 3. This is again the desired behavior for a spiking neural network. Although there are seemingly more spikes for a shorter gap between the spikes, this is not always the case and depends on the internal dynamics of the neural network being stimulated.
CHAPTER 5. IMPLEMENTATION

Figure 5.6: Input Spike trains containing 2 spikes

Figure 5.7: Output Spike trains generated by 2 input spike trains
Case 4 is important because it confirms the claims made in the earlier test; although the stimulation is done 100ms later, the output pattern of cases 2 and 4 are the same. This shows that the neural network is reacting to the relative timing of the input spikes.

5.7.3 Reaction to 3 spikes

In this test all four input patterns are different (figure 5.7), and so are the output patterns (figure 5.8), which is to be expected. Cases 2 and 3 demonstrate that although the input for these cases both have the same firing rate, the generated output are still different from each other, as are their firing rates (even if not by much). This fact is in agreement with the spiking model approach to the simulation of neural networks.

![Figure 5.8: Input Spike trains containing 3 spikes](image)

5.7.4 Larger Input

The following are some examples of input and output spike trains generated using the implementation.

The next set of spike-trains (figure 5.9) show the impact a second input can have on the output of a network. In this figure, the top spike-train is the input, the second is when only the first input neuron fires, the third when only the second input neuron fires and the last one when both of the neurons fire using the same spike-train. Although the second input has little effect on its own, when combined it’s influence can be seen in the final output spike train. The next two figures (5.10 and 5.11) were generated using a very simple network consisting of only 2 input neurons, one core neuron, and one output neuron. An output spike will only be generated if each of the input neurons fire within a period of about 1 ms.
Figure 5.9: Output Spike trains generated by 3 input spikes

Figure 5.10: Effects of a second input neuron
Figure 5.11: 10 Random Spikes

Figure 5.12: 50 Random Spikes
5.7.5 A simple network

The following tests look at the firings of the neurons in the core part of a neural net. The network used is composed of three core neurons, with an input neuron attached to each one. There is also one output neuron, attached to all three of the core neurons. The connections between core neurons have a weight of 1 and a delay of 10 ms. The connections from the input neurons have a weight of 2 and a delay of 0 ms. The connections from the neurons to the output neuron have a weight of 1 and a delay of 0 ms. The thresholds of the core neurons are all set to 1.7. The threshold of the output neuron is set to 2.7.

When the 3 input neurons fire at the same time, the network enters a cyclic behavior, with a frequency of approximately 100 Hz (figure 5.12). Also, this behavior does not appear with only 2 inputs firing at the same time, but if the same 2 inputs are fired a second time at the right moment, it can create the cyclic behavior (figure 5.13). This last example showed the 3 core neurons drift to synchronised firing times, although the drift is hardly visible. To make this behavior more apparent, the thresholds of the core neurons were lowered to 1.1, and each input neuron fired at a different time from the others (0, 1.5 and 2.8). The spike trains of the core neurons clearly show...
CHAPTER 5. IMPLEMENTATION

Figure 5.14: A cyclic firing pattern in a simple network

Figure 5.15: A cyclic firing pattern from 2 inputs
their firings synchronize properly after 4 cycles (figure 5.14). An interesting

Figure 5.16: Synchronized firing in a simple network

property is that the cycles can be turned off with a single well placed spike. In the following example the thresholds of the core neurons are again set to 1.7. All three input neurons are fired at time $t=0$, and one of the inputs is fired again at time $t=30$. This last firing disrupts the end of a cycle in such a way that it is not repeated (figure 5.15). Another property becomes

Figure 5.17: Stopping the repetition in a simple network

apparent when firing the input neurons in between the cyclic firings of the core neurons. The delay of the connections between core neurons has been increased to 15ms. The period of the pattern is now roughly 15.5ms. If the input neurons are then fired in between two firings of the pattern, a second cycle is created, again with a period of roughly 15.5ms. These two cycles affect each other, and drift until seemingly creating a firing pattern with a
period of roughly 7.5ms. With this network, it is possible to have 3 such cycles, with an apparent period of 5 ms. It is possible to stop the patterns using an input spike, but a spike is required for each of the cycles, showing that although they influence each other, they are still independent to a large extent (figure 5.16). These various tests show that the internal dynamics of a spiking neural network can quickly become very complicated as well as have a broad range of firing patterns.
Chapter 6

Conclusion

Rapidly estimating solutions to complex equations through Interval Arithmetic is quite feasible using the algorithm developed. Generally it converges on the solution at a very fast rate. Although the given solution is not always precise. To take full advantage of Interval Arithmetic would require a different solving technique to be used in conjunction with the algorithm presented, to compensate for errors in the calculated range.

The Spiking Response Model, with it’s truly flexible and very broad functionality, is ideal for a neural network consisting of neurons with symbolic internal states. The real advantage of such an internal state is that to estimate future behaviour consists in solving equations quickly. Combining all of these tools allows for a much easier transition from a discrete time simulation to an event based simulation. Also, the overhead of managing events is quite small when using a heap. Such a system works, and the neural networks simulated demonstrate the kinds of behaviour that is expected of a spiking neural network.

Unfortunately, the system developed didn’t include a training ability, but it still shows that it is possible to use Interval Arithmetic to estimate solutions to equations. It also shows that spiking neurons with symbolic internal states are well suited to use in an event based simulation, and that for such a simulation a heap is very suitable for managing the generated events.
Bibliography


Appendix A

Source Code from Simon Langley

This is the original source code sent by Simon Langley. It allowed for the basic integration of Interval Arithmetic into GiNaC. The code used in the project for this purpose is an extended version of this code.
A.1 ivl.cpp

```cpp
#include <ivl.h>
#include <cln/float.h>
#include <stdexcept>
#include <iostream>

/* ---------------------------- rounding GiNaC numerics */
/*
 * round rounds GiNaC numerics up/down for interval operations. If the numbers are rational no rounding is done.

 Arithmetic is done with the current setting of Digits but to save recomputing m/c epsilon each time the values are cached and recomputed only when the value of Digits changes.
 */

using namespace cln;

numeric ivl::round(const numeric &n, bool up)
    // cached values
    static int oldDigits = 0; // Digits last used in arithmetic
    static clF up, down, negEpsilon, epsilon;
    if(n.is_rational()) return n;
    // check Digits setting
    if(Digits != oldDigits)
        { float formatDate = floatFormat(oldDigits = Digits);
          clF one = clF(1, save);
          up = one + floatEpsilon(save);
          down = one - floatNegativeEpsilon(save);
          negEpsilon = leastNegative(float(save));
          epsilon = leastPositive(float(save));
        }
    // convert number to clN format
    clF xf = the<clF>(n.to<clN>());
    if(up) // round up
        { if(xf > 0.0)
            xf = xf + up;
          else if(xf < 0.0)
            xf = xf - down;
        }
    else
        { xf = negEpsilon;
          return numeric(xf);
        }
    return roundRational(n, bool up) // round a rational
    }

static unsigned kludge()
    // a popular trick from long ago
    // we'd like Digits to be initially set large enough that
    // h/w fp isn't used
    // since we can't do that directly we make it a side effect
    // of an otherwise pointless call
    return (Digits = 25);

static unsigned kludge2 = kludge();

const ivl ivl::one(1,1), ivl::zero(0,0);

ivl& ivl::hull(const numeric &n, const numeric &m)
    { if(n > m)
        { sup = m;
          up = n;
        }
    else
        { inf = n;
          up = m;
        }
    return *this;
    }

ivl& ivl::operator *(const ivl &a)
    { numeric ts, ti,
      as, ai;
    bool negit = false;
    if(up < 0)
        { negit = true;    // we so reverse and invert later
          xs = 1.0 - xs;
        }
    } else if(xf > 0.0)
        xf = xf + up;
    else if(xf < 0.0)
        xf = xf - down;
    else
        xf = negEpsilon;
    return numeric(xf);
}
```
APPENDIX A. SOURCE CODE FROM SIMON LANGLEY

```cpp
// Example code

int main() {  
  // Sample use of the code
  
  return 0;
}

if (a.sign() == 0) throw std::domain_error("ivl::operator/ possible division by zero");

if (a.sign() == 0) throw std::domain_error("ivl::pow pow(0,0) undefined");

if (b.sign() == 0) throw std::domain_error("ivl::operator/ possible division by zero");

if (n == 1) return 1;
if (n == 0) {  
  return 1;
  // std::cout << a"^b = (1/a)^b
  // tr = num1/tr;
  n = n;
}

if (n < 0) {  
  // a"^b = (1/a)^(-b)
  n = -n;
  
  if (n < 0) {  
    // std::cout << "a"^b = (1/a)^(-b)
    // tr = num1/tr;
    n = n;
  }
}
```

### Example Output

```
ivl ivl::operator/(const ivl& a) const  
  {  
    if (a.sign() == 0) throw std::domain_error("ivl::operator/ possible division by zero");  
    return operator*(ivl(1/a._up, 1/a._inf, ivl::Rounded));  
  }

ivl & ivl::operator/=(const ivl & a)  
  {  
    if (a.sign() == 0) throw std::domain_error("ivl::operator/ possible division by zero");  
    return operator*=(ivl(1/a._up, 1/a._inf, ivl::Rounded));  
  }
```

### Example Usage

```cpp
// Use of the code

int main() {  
  // Sample use of the code
  
  return 0;
}
```
tr.negate(); // (z)\textsuperscript{2}n = z\textsuperscript{2}n
else if(s < 0)
{
    t = ivl::one;
    while(n > 0) // usual power loop to do a, b as a pair
    {
        if((n & 1) == 1)
            t *= tr;
        if(--n == 0) break;
        n = n >> 1;
    }
    return t;
}
ivl ivl::toFloat() const
{
    return ivl(roundRational(inf, false), roundRational(up, true));
}
ivl& ivl::asFloat()
{
    up = roundRational(up, true);
    inf = roundRational(inf, false);
    return *this;
}
ivl ivl::convert(const ex& e)
{
    if(is_a<numeric>(e)) return ivl(ex_to<numeric>(e));
    if(is_a<add>(e)) return e_to<ivl>(e);
    unsigned n = e.nops();
    if(is_a<add>(e))
    {
        ivl t(0);
        for(unsigned i = 0; i < n; i++)
            t += ivl::convert(e.op(i));
        return t;
    }
    if(is_a<mul>(e))
    {
        ivl t(1);
        for(unsigned i = 0; i < n; i++)
            t *= ivl::convert(e.op(i));
        return t;
    }
    if(is_a<power>(e))
    {
        ivl t(1);
        int n = e.to<numeric>().pow(e.to<numeric>(e.op(1)).to<int>(0));
        throw std::domain_error("ivl: from ex unrecognised operation");
    }
    bool ivl::intersects(const ivl& a) const
    {
        const numeric ka = a.up,
        &ai = a.inf;
        return (((inf <= ka) && (inf >= ai))
            ||((up <= ka) && (up >= ai)));
    }
    ivl ivl::operator | (const ivl& a) const
    {
        if(intersects(a))
            return max(a);
        else
            throw std::domain_error("ivl: union of disjoint ivls");
    }
    ivl& ivl::operator | (const ivl& a)
    {
        if(intersects(a))
            return maxWith(a);
        else
            throw std::domain_error("ivl: union of disjoint ivls");
    }
    ivl ivl::operator & (const ivl& a) const
    {
        if(intersects(a))
            return min(a);
        else
            throw std::domain_error("ivl: intersection of disjoint ivls");
    }
    ivl& ivl::operator & (const ivl& a)
    {
        if(intersects(a))
            return minWith(a);
        else
            throw std::domain_error("ivl: intersection of disjoint ivls");
    }
    // GiNaC methods
    GINAC_IMPLEMENT_REGISTERED_CLASS(ivl, basic)
    const unsigned TINFO_ivl = 0x42430002U;
\_inf = \_other \_inf;

void ivl::copy(const ivl &\_other)
{ inherited::copy(\_other);

\_sup = \_other \_sup;
}

void ivl::destroy(bool call\_parent)
{ if(\_call\_parent) inherited::destroy(\_call\_parent);

//Never returns 0 as A - A ! 0
//better would be sign(this - \_other) but 0 would be a problem!
int ivl::compare\_same\_type(const GiNaC::basic &\_other)
{ const ivl &\_io = static\_cast<const ivl &\_other>;
  return (\_inf >\_inf) ? 1 : (\_sup >\_sup) ? 1 : -1;
}

ivl::ivl(const GiNaC::archive\_node &n, GiNaC::lst &sym\_lst)
{ inherited(n, sym\_lst);
  ex ret;
  n.find\_ex("inf", ret, sym\_lst);
  \_inf = ex\_to\_numeric<ret>(ret);
  n.find\_ex("sup", ret, sym\_lst);
  \_sup = ex\_to\_numeric<ret>(ret);
}

void ivl::archive(GiNaC::archive\_node &n) const
{ inherited::archive(n);
  n.add\_ex("inf", \_inf);
  n.add\_ex("sup", \_sup);
}

GiNaC::ex ivl::unarchive(const GiNaC::archive\_node &n, GiNaC::lst
&sym\_lst)
{ return (new ivl(n, sym\_lst))->set\_flag(status\_flags::dyn\_allocated);
}

void ivl::print(const print\_context& c, unsigned level) const
{ c.s << (*this);
}

bool ivl::info(unsigned inf) const
{
  return \_inf.info(inf) && \_sup.info(inf);
}

size\_t ivl::nops() const
{ return 2;
}

ex ivl::op(size\_t i) const
{ if(i == 0) return \_inf;
  if(i == 1) return \_sup;
  throw std::domain\_error("ivl:op\_parameter not 0 or 1");
}

ex ivl::eval\_int(level = 0) const
{ return middle();
}

A.2 ivl.h

#ifndef ivl\_h
#define ivl\_h

#include <ginac/ginac.h>

using namespace GiNaC;

//if you are already including max/min from std template library
//these should probably become
//
//template<>
//inline max\_numeric\<\_const numeric\& n, const numeric\& m>
//... etc
//and likewise for min

inline numeric max(\_const numeric\& n, \_const numeric\& m)
{ return (n >\_m) ? n : m;
}

inline numeric min(\_const numeric\& n, \_const numeric\& m)
{ return (n <\_m) ? n : m;
}

// declare ivl class number for GiNaC
extern const unsigned TINFO\_ivl;

class ivl : public basic
{
// make it part of GiNaC

GINAC_DECLARE_REGISTERED_CLASS(ivl, basic)

class Rounding { // just to give a name for when rounding is required
protected:
    numeric _inf, _sup;

public:
    // round down/up float values
    static numeric round(const numeric & n, bool up);
    static numeric roundRational(const numeric & n, bool up);
    static Rounding Rounded;
    static const ivl one, zero;

    // two constructors are declared by GiNaC
    // ivl() : _inf(0), _sup(0) {}
    // ivl(const ivl & n) :
    //     _inf(n._inf), _sup(n._sup) {}
    explicit ivl(const numeric & n) :
        inherited(TINFO_ivl), _inf(n), _sup(n) {};

    ivl(const numeric & n, Rounding) :
        inherited(TINFO_ivl), _inf(ivl::round(n, false)), _sup(ivl::round(n, true)) {};

    ivl(const numeric & n, const numeric& m) :
        inherited(TINFO_ivl), _inf(n), _sup(m) {};

    ivl(const numeric & n, const numeric& m, Rounding) :
        inherited(TINFO_ivl), _inf(ivl::round(n, false)), _sup(ivl::round(m, true)) {};

    ivl& operator = (const numeric & a) {
        ivl &op = {ivl(a);
            return *this;
        }

    ivl & widen(const numeric & n, const numeric & m) {
        _inf = ivl::round(n, false);
        _sup = ivl::round(m, true);
        return *this;
    }

    ivl & widen() {
        _inf = ivl::round(_inf, false);
        _sup = ivl::round(_sup, true);
        return *this;
    }

    ivl & hull(const numeric & n, const numeric & m) {
        ivl &hull() {
            if (_inf > _sup) {
                numeric t = _inf;
                _inf = _sup;
                _sup = t;
            }
            return *this;
        }

    ivl & set(const numeric & n, const numeric & m) {
        _inf = n; _sup = m;
        return *this;
    }

    // arithmetic
    ivl operator + (const ivl & a) const
        { return ivl(_inf + a._inf, _sup + a._sup, Rounded); }

    ivl operator + (const numeric & a) const
        { return ivl(_inf + a, _sup + a, Rounded); }

    ivl & operator += (const ivl & a) :
        { return widen(_inf + a._inf, _sup + a._sup); }

    ivl & operator += (const numeric & a) :
        { return widen(_inf + a, _sup + a); }

    ivl operator - (const ivl & a) const
        { return ivl(_inf - a._inf, _sup - a._sup, Rounded); }

    ivl operator - (const numeric & a) const
        { return ivl(_inf - a, _sup - a, Rounded); }

    }
```cpp
// interface
const numeric& inf() const { return _inf; }
const numeric& sup() const { return _sup; }
static ivl convert(const ex& e);
// float endpoints making a new interval
ivl toFloat() const;
// float endpoints in place
ivl& asFloat();

// intersection etc
bool intersects(const ivl& const; ivl& operator &= (const ivl&); ivl& operator | (const ivl&); ivl& operator = (const ivl&);

// max = union if intervals definitely overlap
ivl max(const ivl& a) const
{ return ivl(_min(_inf, a._inf), _max(_sup, a._sup)); }
ivl& maxWith(const ivl& a)
{ _inf = _min(_inf, a._inf); _sup = _max(_sup, a._sup); return *this; }

// min = intersection if ivls definitely overlap
ivl min(const ivl& a) const
{ return ivl(_max(_inf, a._inf), _min(_sup, a._sup)); }
ivl& minWith(const ivl& a)
{ _inf = _max(_inf, a._inf); _sup = _min(_sup, a._sup); return *this; }

// ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
numeric width() const
{ return ivl::round(_sup - _inf, true); }
numeric middle() const
{ return (_inf + _sup)/2; }
ex evaflf() const { return middle().evaflf(); }
numeric radius() const
{ return ivl::round(((_sup - _inf)/2), true); }
numeric abs() const
{ return max(GiNaC::abs(_inf), GiNaC::abs(_sup)); }
// mig = smallest in abs magnitude
numeric mig() const
```
void print(const print::context & c, unsigned level = 0) const {
    bool info(unsigned inf) const;
    size_t nops() const;
    ex op(size_t i) const;
    ex eval(const numeric & n) const;
}

// Inherited from GiNaC::basic and overridden
// I/O for GiNaC

inline std::ostream& operator << (std::ostream& os, const ivl& iv)
{
    GiNaC::operator <<(os, static_cast<const numeric&>(iv.inf()));
    os << ". . .
    GiNaC::operator <<(os, static_cast<const numeric&>(iv.sup()));
    return os;
}

inline ivl operator + (const numeric& a, const ivl& b) {
    return b+a;
}

inline ivl operator - (const numeric& a, const ivl& b) {
    return ivl(a) - b;
}

inline ivl operator * (const numeric& a, const ivl& b) {
    return b * a;
}

extern ivl operator / (const numeric&, const ivl&);

inline ivl pow(const ivl& r, int n) {
    return r.pow(n);
}

inline ivl pow(const ivl& r, const numeric& n) {
    return r.pow(n.to<int>());
}

// Relational operators

// treated as inclusion – not can throw error for overlap
inline bool operator == (const ivl& a, const ivl& b) {
    return a.inf() == b.inf() && a.sup() == b.sup();
}

inline bool operator == (const numeric& a, const ivl& b) {
    return a == b.inf() && a <= b.sup();
}

inline bool operator == (const ivl& a, const numeric& b) {
    return b == a;
}

inline bool operator != (const ivl& a, const ivl& b) {
    return ! (a == b);
}

inline bool operator != (const numeric& a, const ivl& b) {
    return ! (a == b);
}

inline bool operator != (const ivl& a, const numeric& b) {
    return ! (b == a);
}

inline bool operator < (const ivl& a, const ivl& b) {
    return (a.inf() > b.inf()) && (a.sup() < b.sup());
}

inline bool operator < (const numeric& a, const ivl& b) {
    return (a >= b.inf()) && (a <= b.sup());
}

inline bool operator <= (const numeric& a, const ivl& b) {
    return (a >= b.inf()) && (a < b.sup());
}

inline bool operator >= (const ivl& a, const ivl& b) {
    return b < a;
}

inline bool operator > (const ivl& a, const ivl& b) {
    return a < b;
}
{ return b <= a;
inline bool operator >= (const ivl& a, const ivl& b)
{
    return a < b;
}
inline bool operator >= (const ivl& b, const numeric& a)
{ return a < b;
}
}
/** Specialization of is exactly a<ivl>(obj) for symbol objects. */
template<>
inline bool is exactly a<ivl>(const basic & obj)
{
    return obj.tinfo() == TINFO<ivl>;
}
#endif

// function versions of member functions
inline int sign(const ivl& iv) { return iv.sign();
inline const numeric& inf(const ivl& iv) { return iv.inf();
inline const numeric& sup(const ivl& iv) { return iv.sup();
inline numeric width(const ivl& iv) { return iv.width();
inline numeric middle(const ivl& iv) { return iv.middle();
inline numeric radius(const ivl& iv) { return iv.radius();
inline numeric abs(const ivl& iv) { return iv.abs();
inline numeric mig(const ivl& iv) { return iv.mig();
inline ivl toFloat(const ivl& iv) { return iv.toFloat();
}
Appendix B

Project Source Code

This is part of the sourcecode developed for the project. Included is the code relevant to a neuron object and the code for managing a heap.
# B.1 Neuron.cpp

```cpp
#include "Neuron.h"

/* define all the stuff for a basic neuron */
Neuron::Neuron()
{
    next_firing = -1000;
    last_firing = -1000;
}

Neuron::Neuron(NeuralNet * net)
{
    internalState = 0;
    neighborhood = net;
    threshold = 2;
    inhibitory = false;
    debug = false;
    timeStamp = 0;
    next_firing = -1000;
    last_firing = -1000;
    theLinks = NULL;
    theLinksLength = 0;
    // pthread_mutex_init(&theLock, NULL);
}

Neuron::Neuron(NeuralNet * net, numeric thres)
{
    internalState = 0;
    neighborhood = net;
    threshold = thres;
    inhibitory = false;
    debug = false;
    timeStamp = 0;
    next_firing = -1000;
    last_firing = -1000;
    theLinks = NULL;
    theLinksLength = 0;
    // pthread_mutex_init(&theLock, NULL);
}

Neuron::Neuron()
{
    delete [] theLinks;
    // pthread_mutex_destroy(&theLock);
}

void Neuron::setInhibitory(bool value)
{
    inhibitory = value;
}

void Neuron::addConnection(Connection * con)
{
    // cout << "connection added to " << this << endl;
    // create a set of connections that have 1 extra space
    Connection * temp = new Connection *[theLinksLength + 1];
    // copy over the old stuff
    for (int i = 0; i < theLinksLength; i++)
    {
        temp[i] = theLinks[i];
    }
    delete theLinks;
    theLinks = temp;
    theLinks[theLinksLength] = con;
    theLinksLength++;
    return;
}

void Neuron::addPSP(ex in, numeric time, numeric length)
{
    // try removing redundant data everytime it's added ;)
    // ex temp = 0;
    numeric simtime = neighborhood->currentTime;
    // ensure that it doesn't brake with float underflow...
    if (simtime > (lastFiring + 3))
    {
        ivl values;
        if (ivl < add > (internalState))
        {
            for (int i = 0; i < internalState.nops(); i++)
            {
                values;
                ivl::convert((internalState.op(i)).subs(t =
                
                if ((values.sup() < neighborhood->noiseLevel)
```
APPENDIX B. PROJECT SOURCE CODE

```cpp
{  
    && (values.inf() > (-1 * neighborHood->noiseLevel))
}

//
if (debug)
{
    cout << "Removing" << internalState.op(i) << endl;
} // don't do anything... */
else
{
    temp += internalState.op(i);
}
internalState = temp;
}

// first priority is to add the pop::
internalState = internalState + i;

//
if (debug)
{
    cout << "CurrentTime is" << neighborHood->currentTime << endl;
    cout << "Adding PSP to" << this << " who has threshold" << threshold << endl;
    cout << "lastFiring" << lastFiring << endl << "nextFiring" << nextFiring << endl;
    cout << "InternalState is" << care_float << this->internalState << dflt << endl;
}

// check the firing time
// detectFiringTime(time, length);
// detectFiringTime(time, time + neighborHood->delayMax + 10);
}

void Neuron::detectFiringTime (numeric time, numeric length)
{
    // now check for next firing time:
    // interval to check should be from "time" to "length"
    bool foundtime = false;
    stack < numeric >thestack;
    if (next_firing > time && next_firing <= last_firing)
    {
        thestack.push(time + length);
        thestack.push(time);
    }
    int counter = 0;
    numeric new_firing = -1000;
    if (debug)
    {
        cout << "InternalState is" << this->internalState << endl;
        cout << "Detecting time" << endl;
    }
    // while (thestack.size() != 1)
    while (thestack.size() != 1 && !thestack.empty())
    {
        counter++;
        numeric lowerboundary = thestack.top();
        // pop the lower boundary
        thestack.pop();
        numeric upperboundary = thestack.top();
        // don't pop upper boundary
        thestack.pop();
        //
        if (debug)
        {
            cout << "Last fired: " << last_firing << endl;
            cout << lowerboundary << endl;
            cout << upperboundary << endl;
            cout << "i" << endl;
        } //
        //
        ivl values = ivl::convert (internalState.subs (lst (t == ivl (lowerboundary, upperboundary)),
        l ==
        (ivl (lowerboundary, upperboundary) - ivl (last_firing)) \}));
```
APPENDIX B. PROJECT SOURCE CODE

```cpp
//
if (debug)
{
    cout << "Range: [ " << values.inf() << " : " << values.sup() << "] " << endl;
    //cout<<values.inf()<<endl;
    //cout<<values.sup()<<endl;
}

if (values.sup() > threshold)
{
    //check for accurate precision here, 0.01 seems ok to me
    if ((upperboundary - lowerboundary) < neighborHood->precision)
    {
        new_firing = upperboundary;
        foundtime = true;
        break;
    }
    //not at required precision yet, so just add
    //two subintervals to the stack
    //theStack.push(upperboundary);
    //if the second interval is already split, might gain
    //on average seek time, maybe experiment later
    //theStack.push((upperboundary + lowerboundary) / 4)
    //theStack.push((upperboundary + lowerboundary) / 4)
    //theStack.push((upperboundary + lowerboundary) / 2);
    theStack.push((upperboundary + lowerboundary) / 2);
    theStack.push(lowerboundary);
}

//it's not positive, so don't do anything

//
if (debug)
{
    cout << counter << ", iterations to determine firing time: " << endl;
    //we should update the heap with a new event, and our timestamp
    //is guaranteed individual for each event per neuron ;)
    //ensure that a firing time was found, and that it is valid;
    //previous estimation isn't earlier than tested interval.
    //previous estimation isn't earlier than last firing.
    if (foundtime
        && (next_firing > new_firing) || (next_firing <= last_firing))
    {
        //perform some nasty rounding :)
        // doesn't do jack... ;)
        cout << new_firing << endl;
        new_firing = new_firing + neighborHood->precision * 5;
        cout << new_firing << endl;
        new_firing = new_firing / neighborHood->precision / 10;
        cout << new_firing.integer_content() << endl;
        new_firing = new_firing.integer_content() + neighborHood->precision / 10;
        cout << new_firing << endl;
        //
    }

    next_firing = new_firing;
    timeStamp += 1;
    if (debug)
    {
        cout << "new Event timestamp " << timeStamp << " firing at " << next_firing << endl;
    }
}

(neighborHood->theHeap)->
    addEvent (new Event (timeStamp, next_firing, this));
if (debug)
{
    cout << next_firing;
}

// didn't find a time, but old one was erased
//make sure previous firing time was in the interval.
else if ((next_firing >= time) && (next_firing <= (time + length))
{
// could probably only check small interval, and if given
```
APPENDIX B. PROJECT SOURCE CODE

by all that much unless there is a huge maxDelay.
could check larger internal, could speed things up, but I
don't think

*/

nextFiring = lastFiring;
timestamp += 1;
if (debug)
{
cout << "For" << this << endl;
}

void
Neuron::fire()
{
if (debug)
{
cout << "Firing neuron" << this << " at " << neighborhood->
currentTime << endl;
}
lastFiring = neighborhood->currentTime;
//We want to remove any psp's that don't affect the neuron
anymore

ex temp = 0;
ivl values;
if (is<add>(internalState))
{
    for (int i = 0; i < internalState.nops(); i++)
    {
        //Set start to be 1 time unit after firing.
        values =
            ivl::convert(internalState.op(i)).
            sub(1st
                (t =
                    ivl(lastFiring + 3,
                        lastFiring + neighborhood->
                        delayMax +
                        10), 1 = lastFiring));
        if ((values.sup() < neighborhood->noiseLevel)

            & & (values.inf() > (-1 + neighborhood->noiseLevel))
        }
        if (debug)
        {
            cout << "Removing" << internalState.op(i) << endl;
        } //Don't do anything.
        else
        {
            temp += internalState.op(i);
        }
    }
}
internalState = temp;

//*/
//numeric fire_time = next_firing;
//Place reset kernel at firing time
//internalState += ResetKernel.subs(t = t - fire_time);
if (debug)
{
cout << "Adding ResetKernel" << csrc float << ResetKernel
    .subs(t =
        -
        lastFiring
    ) << dflt << endl;
}
internalState = internalState + ResetKernel.subs(t =
    t -
    lastFiring);
//cout << "ResetKernel.subs(t = t - next_firing)" << endl;
/*
detect next firing time
for length, use max delay + a generous offset for a psp
because if ipsps, and new firing time outside scope of ipsps,
wouldn't find it properly, although see note in
detectFiringTime

/ *
if (debug)
{
    cout << "going to detect firing time" << endl;
}
*/
```cpp
// detectFiringTime (last_firing + 3, neighborHood->delayMax + 10);
/*
  // fire on all of the connections
  for (int i = 0; i < theLinksLength; i++)
  { theLinks[i]->fire();
  } // cout << "finished firing" << this << endl;
*/

void Neuron::setDebug (bool d)
{ debug = d; }

numeric Neuron::getTimeStamp()
{ return timeStamp; }

/* all the definitions for an Output Neuron */
OutputNeuron::OutputNeuron (NeuralNet * net, numeric thres)
{ /* this is just neuron stuff */
  internalState = 0;
  neighborHood = net;
  threshold = thres;
  timeStamp = 0;
  nextFiring = -1000;
  lastFiring = -1000;
  theLinks = NULL;
  theLinksLength = 0;
  inhibitory = false;
  debug = false;
  /* the output stuff */
  theInputs = NULL;
  theInputsLength = 0;
  fireTimes = new vector<numeric> ();
}

OutputNeuron::~OutputNeuron ()
{ delete [] theLinks; }

void OutputNeuron::fire (void)
{ if (debug)
  { cout << "Firing output neuron" << this << endl;
  }

  lastFiring = neighborHood->currentTime;
  // we want to remove any pps that don't affect the neuron anymore
  ivl values;
  ex temp = 0;
  if (is_a < add > (internalState))
  { for (int i = 0; i < internalState.nops(); i++)
    { // set values to be 1 ms after firing
      values = ivl::convert *((internalState.firing(i));
      subs (lat
          svn = (last_firing + 3,
                 last_firing + neighborHood->delayMax +
                 10), l = last_firing));

      if ((values.sup() < 0.1) && (values.inf() > -0.1))
      { // don't do anything...
      }
      else
      { temp += internalState.firing(i);
      } } // place reset kernel at firing time
  interalState += ResetKernel.subs (t = t last_firing);
  // add to the list of times when neuron was fired
  fireTimes->push_back (last_firing);
  /* detect next firing time, not sure about length, so 1 second...*/
  detectFiringTime (last_firing + 3, neighborHood->delayMax + 10);
  */
```
void
OutputNeuron::addInputConnection (Connection *con)
{
    // create a set of connections that have 1 extra space
    Connection *temp = new Connection [*theInputsLength + 1];
    // copy over the old stuff
    for (int i = 0; i < theInputsLength; i++)
    {
        temp[i] = theInputs[i];
    }
    delete theInputs;
    theInputs = temp;
    theInputs[*theInputsLength] = con;
    theInputsLength++;
    return;
}
vector < numeric > *OutputNeuron::getFireTimes (void)
{
    return fireTimes;
}
InputNeuron::InputNeuron (NeuralNet *net)
{
    internalState = 0;
    neighborhood = net;
    threshold = 0;
    timeStamp = 0;
    nextFiring = 0;
    theLinks = NULL;
    theLinksLength = 0;
    inhibitory = false;
    debug = false;
    // pthread_mute(pthread_mutex_t theLock);
    int theLinksLength;
    /*
         void addPSP (ex in, numeric time, numeric length);
         virtual void fire ()
         void addConnection (Connection *con);
         numeric getTimeStamp (void);
         numeric getNextFiring (void);
         void setInhibitory (bool value);
         void setDebug (bool d);
         Neural ();
         Neural (NeuralNet * net);
         Neural (NeuralNet * net, numeric three);
         ~Neuron ();
         void detectFiringTime (numeric time, numeric length);
     */
}
#include "Heap.h"

void Heap::addEvent (Event * ev)
{
    Event * swap;
    fill ++;
    long i = fill;
    // check that the heap isn't full (this shouldn't happen, as size of
    // heap should be set accordingly)
    // put the event in the right place on the heap, starting from
    theEvents[i] = ev;

    Connection * theInputs;
    int theInputsLength;
    vector < numeric > * fireTimes;
    public:
    vector < numeric > * getFireTimes (void);
    void addInputConnection (Connection * con);
    void fire (void);
    OutputNeuron (NeuralNet * net , numeric theRes);
    OutputNeuron ( );
    ~ OutputNeuron ( );
    public:
    ~ InputNeuron ( );
    void fire (void);
    class InputNeuron : public Neuron
    
    class OutputNeuron : public Neuron
    
    #endif

    B.3 Heap.cpp

    #include "Heap.cpp"

    void Heap::addEvent (Event * ev)
    {
        Event * swap;
        fill ++;
        long i = fill;
        // check that the heap isn't full (this shouldn't happen, as size of
        // heap should be set accordingly)
        // put the event in the right place on the heap, starting from
        theEvents[i] = ev;

        Connection * theInputs;
        int theInputsLength;
        vector < numeric > * fireTimes;
        public:
        vector < numeric > * getFireTimes (void);
        void addInputConnection (Connection * con);
        void fire (void);
        OutputNeuron (NeuralNet * net , numeric theRes);
        OutputNeuron ( );
        ~ OutputNeuron ( );
        public:
        ~ InputNeuron ( );
        void fire (void);
        class InputNeuron : public Neuron
        
        class OutputNeuron : public Neuron
        
        #endif

        B.3 Heap.cpp

        #include "Heap.h"

        void Heap::addEvent (Event * ev)
        {
            Event * swap;
            fill ++;
            long i = fill;
            // check that the heap isn't full (this shouldn't happen, as size of
            // heap should be set accordingly)
            // put the event in the right place on the heap, starting from
            theEvents[i] = ev;

            Connection * theInputs;
            int theInputsLength;
            vector < numeric > * fireTimes;
            public:
            vector < numeric > * getFireTimes (void);
            void addInputConnection (Connection * con);
            void fire (void);
            OutputNeuron (NeuralNet * net , numeric theRes);
            OutputNeuron ( );
            ~ OutputNeuron ( );
            public:
            ~ InputNeuron ( );
            void fire (void);
            class InputNeuron : public Neuron
            
            class OutputNeuron : public Neuron
            
            #endif

        / *
        cont << "fill: " << i << endl;
        cont << "New item: " << ev->getFireTime() << " for: ";
        cont << ev->getTheNeuron() << endl;
        *
        while ((i > 1)
            && (theEvents[i / 2]->getFireTime() > theEvents[i]->
                getFireTime ( )))
        {
            // swap them, change i...
            swap = theEvents[i / 2];
            theEvents[i / 2] = theEvents[i];
            theEvents[i] = swap;
            i = i / 2;
        }
        // printHeap ();
        //
        #ifdef _WIN32
        pthread_mutex_unlock(theLock);
        #endif

        void Heap::printHeap (void)
        {
            int levels = (int) (std::log ((float) fill) / std::log (2.0));
            // cont << "levels:" << levels << endl;
            for (int i = 0; i <= levels; i++)
            {
                // cont << "items:" << (int)(std::pow(2,0,i)) << endl;
                for (int j = 0; j < (int) (std::pow (2,0,i)); j++)
                {
                    if (((int) std::pow (2,0,i) + j) > fill)
                    {
                        i = levels;
                        break;
                    }
                    else
                    {
                        cout << theEvents[(int) std::pow (2,0,i) + j]->
                            getFireTime () << endl;
                    }
                    cout << endl;
                }
            }
        }

        void Heap::runEvent (void)
APPENDIX B. PROJECT SOURCE CODE

```cpp
// rearrange the heap after popping the top element
{
    // cout << "starting event" << endl;
    Event *swap;
    pthread_mutex_lock(&theLock);
    // printHeap();
    // put bottom into top
    Event *current = theEvents[1];
    theEvents[1] = theEvents[fill];
    fill --;
    long i = 2;
    while (i <= fill)
    {
        // test left leaf, swap if necessary
        if ((theEvents[i / 2] > getFireTime()) & (theEvents[i] > getFireTime()) & (theEvents[i + 1] > getFireTime()))
            // cout << "left swap" << endl;
            swap = theEvents[i];
            theEvents[i] = theEvents[i / 2];
            theEvents[i / 2] = swap;
            i *= 2;
        // test right leaf, swap if necessary
        else if ((theEvents[i / 2] > getFireTime()) & (theEvents[i] > getFireTime()) & (theEvents[i + 1] > getFireTime()))
            // cout << "right swap" << endl;
            i++;
            swap = theEvents[i];
            theEvents[i] = theEvents[i / 2];
            theEvents[i / 2] = swap;
            i *= 2;
        // in the right place, so exit
        else
            break;
    }
    pthread_mutex_unlock(&theLock);
}
```

```cpp
// run the popped event
if (current->getTimeStamp() == (current->getTheNeuron())->getTimestamp())
{
    // this is a valid event, so trigger the neuron
    // update simulation time
    // if (neighborHood->currentTime > current->getFireTime())
    // {
    //     printHeap();
    //     cout << "mistake somewhere" << endl;
    // } else
    // {
    //     neighborHood->currentTime = current->getFireTime();
    //     cout << "firing" << current->getTheNeuron() << endl;
    //     (current->getTheNeuron())->fire();
    // }
    // printHeap();
    else if (fill > 0)
    {
        runEvent();
    }
}
```

```cpp
Heap::Heap (NeuralNet *net)
{
    fill = 0;
    neighborHood = net;
    theEvents = new Event[MAXEVENTS];
    // cout << "heap 1" << endl;
    // pthread_mutex_init(&theLock, NULL);
    // cout << "heap 2" << endl;
}
```

```cpp
Heap::~Heap (void)
{
    fill = 0;
    neighborHood = net;
    theEvents = new Event[MAXEVENTS];
    // cout << "heap 1" << endl;
    // pthread_mutex_init(&theLock, NULL);
    // cout << "heap 2" << endl;
}
```

```cpp
Heap::nextFireTime (void)
{
    return theEvents[1]--getFireTime();
}
```
B.4 Heap.h

```c
#ifndef HEAP_H
#define HEAP_H

#include "NeuralNet.h"

// simple heap class
class Heap
{
    Event **theEvents;
    NeuralNet *neighborhood;
    // long fill;
    // pthread_mutex_t theLock;
    public:
    long fill;
    Heap (NeuralNet *net);
    ~Heap ();
    void addEvent (Event *ev);
    void runEvent (void);
    numeric getNextFireTime (void);
    void printHeap (void);
};
#endif
```