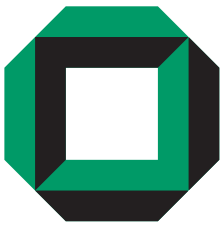

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Technische Informatik
Lehrstuhl Prof. Dr. J. Henkel

Study Thesis

Yinspire – A performance efficient simulator for spiking neural nets

Michael Neumann
mneumann@ntecs.de

Nov. 2007 - May 2008

Supervisors:
Dipl.-Inform. F. Kaiser
Dr. F. Feldbusch

Yinspire – a novel discrete-event and process-oriented simulator for spiking neural nets is presented, which focuses both on performance and on clean, object-oriented design. Performance is mainly achieved by using a *divide-and-conquer* approach for the priority queue that maintains the pending event set, but also by careful implementation. Benchmarks show that Yinspire performs an order of magnitude better than an existing simulator called Inspire[1]. It's clean and concise implementation will hopefully aid further research by serving as a stable foundation to build upon.

Im Rahmen dieser Studienarbeit entstand *Yinspire* – ein neuer in C++ implementierter, ereignisgesteuerter und prozessorientierter Simulator für gepulste Neuronale Netz. Zielsetzung der Arbeit ist es gewesen, eine hohe Performanz bei der Simulation zu erreichen. Daher wurde bei der Implementierung besonderes Augenmerk auf die Performanz sowie auf eine saubere, objekt-orientierte Architektur gelegt. Die hohe Performanz – Messungen bescheinigen Yinspire eine um mindestens eine Grössenordnung höhere Performanz im Vergleich zu dem Simulator Inspire[1] – wird hauptsächlich durch einen *divide-and-conquer* Ansatz erzielt, wobei die globale Prioritätswarteschlange, zuständig für die ereignisgesteuerte Simulation, in mehrere lokale Warteschlangen intelligent aufgeteilt wird. Weiterhin wird die Performanz durch eine saubere Programmierung sowie durch die bewusste Wahl von entsprechenden Konstrukten erhöht.

Die saubere Implementierung von Yinspire wird hoffentlich die zukünftige Forschung an gepulsten Neuronalen Netzen erleichtern und als Grundgerüst für weitere Entwicklungen dienen.

[1] [Feldbusch and Kaiser, 2005] [Weiß, 2005]

I hereby declare that I wrote this thesis myself without sources other than those indicated herein. All parts taken from published and unpublished scripts are indicated as such.

Karlsruhe, 08.05.2008

Michael Neumann

Contents

Nomenclature

- *Discrete Event Simulation* (DES): "In discrete-event simulation, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system. [Wikipedia]"
- *Process-oriented Simulation*: "In process-oriented simulation, the real system is modeled as collection of processes, each competing for the resources of the system. A process management facility allows processes to become active, to operate in the simulated environment, and to eventually terminate. The process management facility is capable of managing many active processes so that they each appear to be active at the same time. This "pseudo parallelism" for simultaneously active processes is a very important feature of process-oriented simulation. [Schwetman, 1990]". In spiking neural nets process-oriented simulation is used for neuron models that need to solve differential equations which describe the course of potential over time.
- *Stepped scheduling*: This term is used in Yinspire as a synonym for process-oriented simulation.
- *Pending Event Set* (PES): In an event-driven simulator, the pending event set contains all future events. Maintaining and scheduling this pending event set is the core task of any event-driven simulator. A commonly used data structure for this purpose is the priority queue.
- *Event*, *Spike*, *Stimulus* and *Stimuli* (plural): They are all used interchangeably throughout the text. An event marks a change of state in the system. In the context of spiking neural nets, we prefer the terms spike and stimulus over the more technical and implementation specific term *event*.
- *Spiking Neural Net* (SNN): Spiking neural nets are an extension of artifical neural nets for the concept of time, leading to a more realistic model of neural simulation.
- *Pulsed Neural Net*: Synonym for spiking neural net.
- *Scheduler*: The part of the event-driven simulator that implements the chronological scheduling of events, or in case of process-oriented simulation, the part which manages the processes. In Yinspire this term is used interchangably with the term simulator, as Yinspire unifies both event-driven and process-oriented simulation.
- *Simulator*: In Yinspire this term is used interchangably with the term scheduler, as both event-driven and process-oriented simulation are unified. Note however that the classes Scheduler and Simulator are not used interchangably in this text!

- *Schedule Entity*: In Yinspire, this refers to an entity that is subject of being scheduled by the scheduler.
- *Neural Entity*: In Yinspire, this refers to an entity that is part of a neural net, for example a neuron or synapse.
- *Inspire*: A simulator for spiking neural nets developed at the University of Karlsruhe [Feldbusch and Kaiser, 2005] [Weiß, 2005].

1 Introduction

Researchers in the field of spiking neural nets and reservoir computing[1] demand for high-performance simulations of neural nets. To cope with this growing demand, *Yinspire* is presented, which is a new implementation of a discrete-event and process-oriented spiking neural net simulator written from scratch in C++. It focuses both on performance and on clean design, achieving both aspects at the same time by using a novel *divide-and-conquer* approach for the priority queue – the heart of almost every discrete-event simulator – which maintains the pending event set. This leads to a much more generalized architecture where each entity in a neural net is responsible for maintaining its own set of events in a local priority queue – an approach resembling more closely the behaviour found in real nature, with a number of advantages:

- Decisions about how and where to store events (or if at all) can be made locally, leading to a whole new area for performance improvements.
- Aggregation of events (possibly adaptive) with same or similar time becomes easy.
- More efficient storage of events.

Yinspire further generalizes the architecture in regard to the following:

- Unified event-driven and process-oriented simulation (no separate base classes).
- Synapses are first-class objects.
- No need for special hebbsch neurons as every neuron can now act as hebbsch neuron.

1.1 Overview

In Chapter 2, the *Architecture and Implementation* of Yinspire is described in detail. Chapter 3 explains how to extend Yinspire for new neuron or synapse models and how to use Yinspire within your own applications. Chapter 4 deals with the *Installation* of Yinspire, i.e. how to compile Yinspire on your system, while Chapter 5 *Interfaces to Foreign Languages* introduces the Ruby and Matlab interfaces to Yinspire by giving examples, but also includes a (boring) function reference. The Chapter 6 about *Prototypes* presents two prototypes that emerged out of this study thesis: An implementation of Yinspire in a combination of Ruby and C++ and a graphical neural net editor. In Chapter 7 *Performance Benchmark*, Yinspire is compared against the old simulator

[1] for example: liquid state machines

Inspire [Feldbusch and Kaiser, 2005] [Weiß, 2005] showing an increase in performance of at least an order of magnitude. Chapter 8 *Benchmarking Priority Queues* further compares various priority queue algorithms using the Classic Hold model [Jones, 1986]. *Performance Tips* are given in Chapter 9, while Chapter 10 *Outlook* lists possible items to be done by future study theses.

2 Architecture and Implementation

2.1 Overview

In Figure 2.1 the overall architecture of Yinspire using UML notation is shown, leaving out a lot of minor details like miscellaneous classes which are not required for simulation. To make it easier to understand, the diagram is logically split into the following four fragments:

- *Scheduling*: Forms the abstract core of an event-driven and process-oriented simulator. Class Scheduler implements the simulator which schedules instances of class ScheduleEntity. See Section 2.2.
- *Neural Core*: Class NeuralEntity extends class ScheduleEntity for operations specific to spiking neural nets and also serves as the base class and interface for all further entities in a neural net. See Section 2.3.
- *Structural entities*: Class NeuralEntity is further subclassed by Neuron and Synapse, adding structure eminent for neurons and synapses: While neurons can have many pre- and post-synapses, synapses must have exactly one pre- and one post-neuron. This is the fragment where all further structural related behaviour in form of base classes should go, for instance the structure of multi-compartment models. See Section 2.4.
- *Models*: Contains all neural-net related classes which implement behaviour, like for example *Neuron_SRM01*, a neuron type that uses the simple response model, or *Synapse_Default*, the most basic implementation of a synapse. To avoid unnecessary repetition, common behaviour is extracted into superclasses like *Neuron_Base* and *Synapse_Base*. See Section 2.5.

This chapter contains further information about various utility classes, like Loaders and Dumpers, which are briefly described in Section 2.6. A description about the Yin file format used by Yinspire as an external notation for neural nets can be found in Section 2.7, converters to and from legacy file formats are described in Section 2.8 and for the command-line interface of Yinspire itself see Section 2.9.

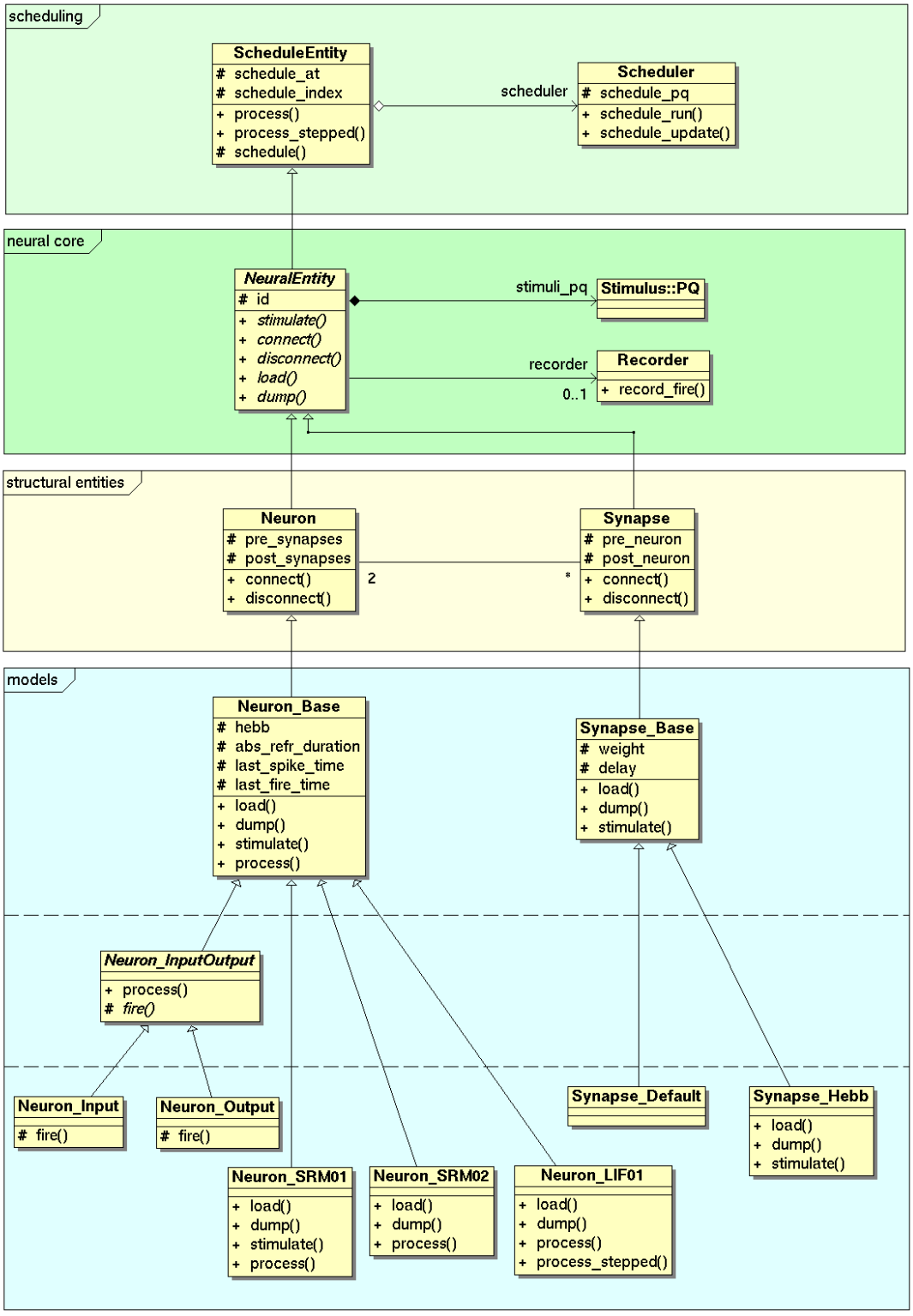

Figure 2.1: Architecture of Yinspire (UML Notation)

2.2 Scheduling

This section describes the part of Yinspire that implements the event-driven and process-oriented simulation.

2.2.1 Introduction

In discrete-time simulation the value of some other magnitude is simulated over time. State changes only at discrete points in time, thus discrete-time simulation. For example in case of a CPU simulator, the simulated value would be the state of the register-file and the status flags, which change as instructions are executed and time progresses. In the case of discrete-time simulation of spiking neural nets, the state that is simulated would be some kind of membrane potential of the neurons, which eventually triggers the firing of neurons when a threshold is reached, which in turn again influences the membrane potential of other neurons and so on. Two approaches come to mind how such a discrete-time simulator could be implemented:

- Given the state of the whole neural net at time t_i, the next state of the whole neural net is calculated for time $t_{i+1} = t_i + \Delta t$, where Δt denotes the smallest possible time unit. This approach is very well suited when a lot of activity happens within a neural net and further allows to calculate the next state of the neural net with the highest possible degree of parallelism. The downside is that the accuracy of the result and the performance depends heavily on the value of Δt.

- The state of an individual neuron is only calculated when something changes that could influence the sate of this neuron. This approach is called *event-driven simulation*. For example, the state is only calculated when a neuron receives a fire impulse through a synapse. For low-activity neural nets this approach is very efficient as the simulation time can skip time intervals where no activiy occurs. Furthermore, the accuracy of time is very precise as it isn't quantified using a Δt. The downsides are, that it is much harder to implement as the events have to be maintained and scheduled for execution using some advanced data structures like priority queues, that achieving parallelism is very hard if not impossible, and that the performance depends heavily on the activity in the neural net.

Yinspire mainly uses the second approach – event-driven simulation – but also allows process-oriented simulation, which is related to the first approach.

2.2.2 Event-driven Simulation

The most basic and common approach to implement an event-driven simulator is to use one single priority queue which contains all events for all entities subject for being simulated – neurons and synapses in case of a neural net. A priority queue data structure is used as it allows for efficient retrieval of elements in sorted order, exactly what is needed for event-driven simulation. In this context, an event usually represents

the firing of a neuron in form of an action potential sent to adjacent entities, while the priority value used for an event in the priority queue represents the timestamp of its appearance, which can be further delayed by the propagation delay of synapses[1]. The priority queue ensures that events are *scheduled* in the order of their appearance[2].

The core routine of such an event-driven simulator – the event loop – is actually quite simplistic and consists of just the following steps:

1. Find the event that occurs next in time.
2. Perform some action for this event. For example, recalculate the state of the neuron this event is targeted at.
3. Remove this event from the priority queue.
4. Repeat with 1. until either the priority queue turns empty or the simulation time reaches a user-specified limit.

Finding the next event in time is efficiently implemented using a priority queue. The *top()* operation of a priority queue returns the element with the highest priority (smallest event time), whereas the *pop()* operation removes that element from the priority queue. To add a new event to the priority queue there is operation *push()*.

2.2.3 Process-oriented Simulation

In contrast to event-driven simulation, process-oriented simulation doesn't involve events at all. Instead, processes are used to model behaviour in a way similar to processes in an operating system or application to model concurrently running activities. Those processes are then triggered at equi-distant time steps Δt, where in general each entity (for example a neuron) uses one process. In the case of spiking neural-nets, process-oriented simulation is used for special neuron models that make use of differential equations to describe the course of their membrane potential. Solving those differential equations numerically requires a method to be invoked at regular time intervals, which is the job of the process scheduler. The same could of course be implemented using an event-driven simulator, though not that efficiently.

One possible implementation of process-oriented simulation is to use two linked lists: One list that contains all active entities and one list that contains the entities that should be deactivated and as such removed from the active list. The simulation time is then increased in equi-distant steps and for each time step all active entities are called to perform some action. Afterwards, all entities contained in the deactivation list are removed from the active list, and the whole procedure is repeated.

Yinspire uses a different, much more efficient approach than the two-list approach to implement process-oriented simulation which will be explained later in Section 2.2.4. Instead of process-oriented simulation we also refer to stepped scheduling as a synonym.

[1] In real neural nets it's the dentrites and axons that delay the signal and not necessarily the synapse. Despite, for simplicity reasons, it is modelled by default this way in Yinspire.

[2] Events are usually scheduled to appear in the future.

2.2.4 Implementation of Yinspire

While the old simulator Inspire stores all events of all entities in one global priority queue, Yinspire only stores the timestamp of the next event for each entity in the global priority queue, whereas the event itself (i.e. timestamp, weight and origin) has to be stored somewhere else. Where exactly the event is stored is the responsibility of the entity that owns the event, i.e. the entity the event is sent to. The default case in Yinspire is that each entity owns a local priority queue into which events get stored. We call this architecture with one global priority queue and many local per-entity priority queues *decentralized scheduling*, which is further depicted in Figure 2.2.

Not shown in Figure 2.2 is that each element of a local priority queue contains a full event (timestamp, weight and origin) whereas an element in the global priority queue only contains the timestamp and a pointer to the entity (the arrow in the figure). The priority of the top element in a local priority queue always equals that of the priority used in the global priority queue for that entity. For example the priority for Neuron1 in the global priority queue is 1.0, which is the same value as the priority of the top element in Neuron1's local priority queue. This condition is always met, i.e. it is invariant. This also means that whenever a local priority queue is modified, for example an event is removed or added, it's entry in the global priority queue has to be potentially updated accordingly. An update in the global priority queue becomes necessary only when the priority of the top-most element of the corresponding local priority queue changes.

One step in the process of scheduling is shown in Figure 2.2 (left hand side vs. right) and is further explained below:

1. Lookup in the global priority queue which entity to schedule next in time. On the left hand side of the figure, this is Neuron1 (timestamp 1.0). It is always the top element of the priority queue.

2. For this entity call method process() with the value of the timestamp as parameter: Neuron1.process(1.0). The timestamp reflects the current simulation time.

3. Method process() now processes all events with that timestamp and removes them from the local priority queue. Usually, the stimuli weight of all events are summed to give the total input stimuli weight, which is then added in some way to the membrane potential of the entity.

4. Removing events from the local priority queue (in our case this is just the event with timestamp 1.0) triggers a reschedule in the global priority queue, as the priority of the top-most element of Neuron1's local priority queue changes from 1.0 to 2.5. See the right hand side of the figure and notice that the new priority of Neuron1 in the global priority queue is now 2.5 and that it is scheduled after Neuron3 (priority 2.3).

5. The same procedure continues with 1.

The global priority queue in Yinspire is maintained by class Scheduler. Each entity in a neural net inherits from class ScheduleEntity, which provides methods related to scheduling. The interface of both classes is shown in Figure 2.3. Each ScheduleEntity contains a reference to the Scheduler. This is required as a ScheduleEntity has to be able to reschedule itself within the global priority queue, which can be accomplished by calling method schedule_update(entity). The priority of each entity in the global priority queue is contained within the instance variable schedule_at of class ScheduleEntity. As such, to schedule itself for a later point in time, a ScheduleEntity has to assign a new value to `schedule_at` and then call `scheduler->schedule_update(this)`. This is exactly how method schedule(at) of class ScheduleEntity is implemented.

Method schedule_run() of class Scheduler contains the main loop that performs the steps explained above, and only returns if the current simulation time exceed the `stop_at` parameter or if the global priority queue turns empty.

The second instance variable of class ScheduleEntity `schedule_index` reflects the position of the ScheduleEntity in the global priority queue. This is required to efficiently modify the priority of a ScheduleEntity in the priority queue, as without knowing the position of the entry whose priority is going to be modified, we would need to traverse the whole priority queue to find the corresponding entry – with $O(n)$ time complexity a costly operation – whereas if we know it's position we can directly start to rebalance from that position and the cost drops down to just $O(\log n)$. As we use an implicit binary heap algorithm for the global priority queue (a complete binary heap laid out in array-form), the position is just the index within the array. You'll never have to modify `schedule_index` yourself as this is the task of the scheduler.

The other instance variables and methods of classes ScheduleEntity and Scheduler (all that contain the word "step") are related to process-oriented simulation, which is also called stepped-scheduling in Yinspire. Process-oriented scheduling in the context of Yinspire means that a ScheduleEntity can request that it's process_stepped() method should be called at regular time intervals. The process_stepped() method will then be called by the Scheduler with the current simulation time and the length of the time interval as parameters. This is independent of method process(), which gets called to process events. The value used as time interval is stored in the `schedule_step` instance variable of class Scheduler.

To request stepped scheduling be activated, a ScheduleEntity inserts itself into a doubly-linked list formed by instance variables `schedule_stepping_list_prev` and `schedule_stepping_list_next`. To disable, it just removes itself from that list. The root of the list is provided by `schedule_stepping_list_root` of class Scheduler. Inserting (enabling) and removing (disabling) from this list is implemented in methods schedule_enable_stepping() and schedule_disable_stepping().

Process-oriented scheduling is used in Yinspire for example by the neuron model Neuron_LIF01, which uses the Leaky-Integrate and Fire model. In Neuron_LIF01, incoming stimuli (i.e. events) activate stepped scheduling, which is then used to compute the course of the membrane potential in method process_stepped() over time (i.e. for each time step) until the membrane potential falls below a given threshold and stepped scheduling is deactivated again. In case the membrane potential exceeds the fire threshold, it will fire and send stimuli to adjacent entities. That is, this neuron

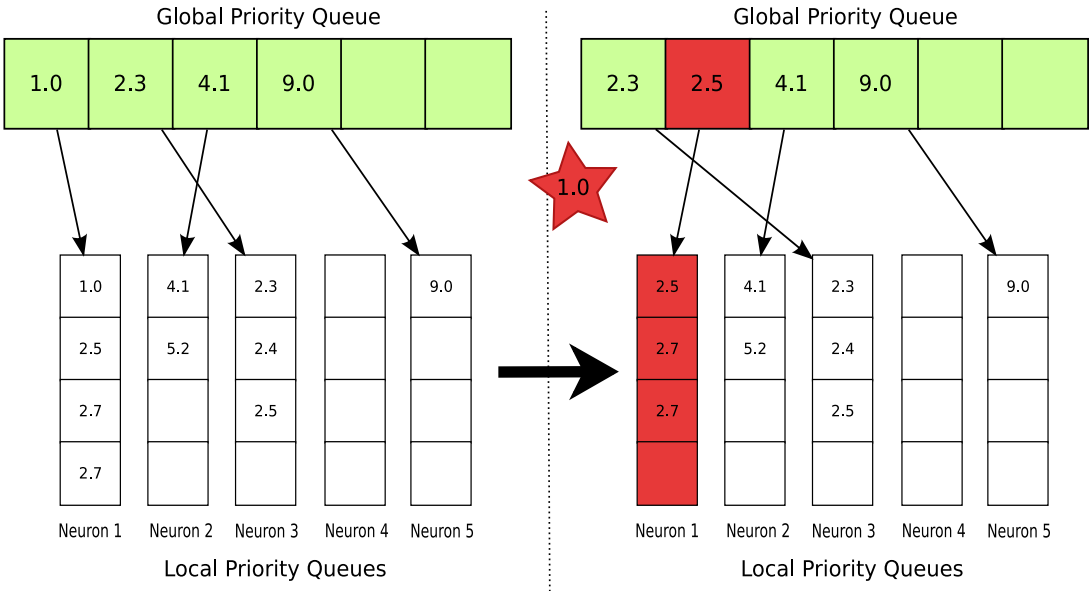

Figure 2.2: Decentralized scheduling architecture of Yinspire

model uses a combination of event-driven and process-oriented simulation. Process-oriented simulation is only used when the neuron is "active". To activate a neuron, events are being used.

Note that the local priority queue is not part of class ScheduleEntity, instead it is part of the direct subclass NeuralEntity. As such, all methods related to how to store an event (and what exactly an event is) is implemented within class NeuralEntity and is explained more detailed in section 2.3. This includes how events are actually sent to other entities, which is not handled in the underlying section. All that scheduling as explained in this section provides is a facility to schedule a ScheduleEntity for being called (method process and/or method process_stepped) at a specific point in time. Everything else is handled by class NeuralEntity and is explained in section 2.3.

Advantages of Decentralized Scheduling

This new decentralizied approach used by Yinspire is in stark contrast with the old simulator Inspire which used *one* global priority queue for *all events* of *all entities*. If you consider that during some simulations a single neuron can have in the range of 10^6 pending events (at a single time), and the neural net itself consist of several hundreds to thousands of neurons, this can lead to a huge priority queue which in turn can have a negative impact on performance.

While performance is likely to increase, splitting – or decentralizing – the priority queue also leads to a much more generalized architecture where each entity is now responsible for maintaining it's own local priority queue. This has a number of advantages:

- Events (stimuli) can be stored more space-efficient (around 30%) in the local

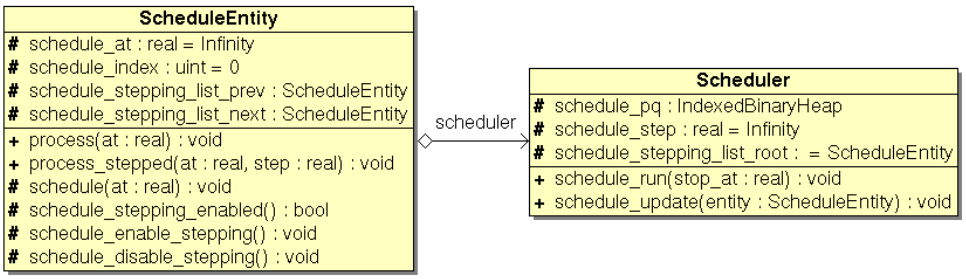

Figure 2.3: Scheduling-related classes

priority queues as a pointer to the entity in each event is avoided.

- It's a lot easier to aggregate events with the same or a similar timestamp.
- Each entity can decide what to do with an incoming event, how to store it or whether to store it at all. For example the priority queue could be made adaptive, aggregating more entries when it grows larger, bounding worst case runtime.
- Some neuron types may want to skip incoming events during their absolute refractory period[3].

Even if the worst-case time complexity for decentralized scheduling still stays the same (see the next section for more information) the flexibility we gain has the potential for a huge increase in performance.

Algorithmic complexity of Decentralized Scheduling

Note that in the following we use a binary heap algorithm for all priority queues and that an access means an insert or delete-min operation. Assuming we have a net with n neurons and that the number of events per neuron is q, then the access time for a unified global priority queue (the "old" approach) is:

$$T_{access_{unified}} = O\left(\log\left(n \times q\right)\right)$$

In the decentralized approach (the "new" approach), the total access time is split into two components: The access time for the local priority queue and the time required to rebalance the global priority queue:

$$T_{access_{local}} = O\left(\log q\right)$$

[3] I measured that this alone doubles performance for the benchmark present in Chapter 7.

$$T_{rebalance_{global}} = O\left(\log n\right)$$

$$T_{access_{decentralized}} = T_{access_{local}} + T_{rebalance_{global}} = O(\log\left(n \times q\right))$$

While a delete-min operation of the local priority queue always triggers a rebalancing of the global priority queue, this is not the case for an insert operation, where a rebalance only happens if the insert would change the earliest timestamp of the local priority queue. Regardless of this fact, when considering worst-case time complexity the decentralized approach doesn't provide an improvement over the unified approach, whereas in practice nearly half of the time there is no need to rebalance the global priority queue. Let the rebalancing of the global priority queue occur with probabiliy p, this gives us:

$$T_{access_{decentralized}}(p) = p \times T_{rebalance_{global}} + T_{access_{local}} = O\left(\log\left(n^p \times q\right)\right)$$

For a rebalancing probability of $p = 0.5$ we get a speedup of the decentralized approach over the unified approach of:

$$S_{0.5} = \frac{\log(nq)}{\log(n^{0.5}q)} \geqslant 1$$

This means that the more neurons we have in our net the higher the gained speedup will be. As an example Figure 2.4 shows the gained speedup for serveral probabilities with a net of n=1000 neurons and a small local priority queue size q. Notice that the real speedup might be different, for example due to improvements in cache locality.

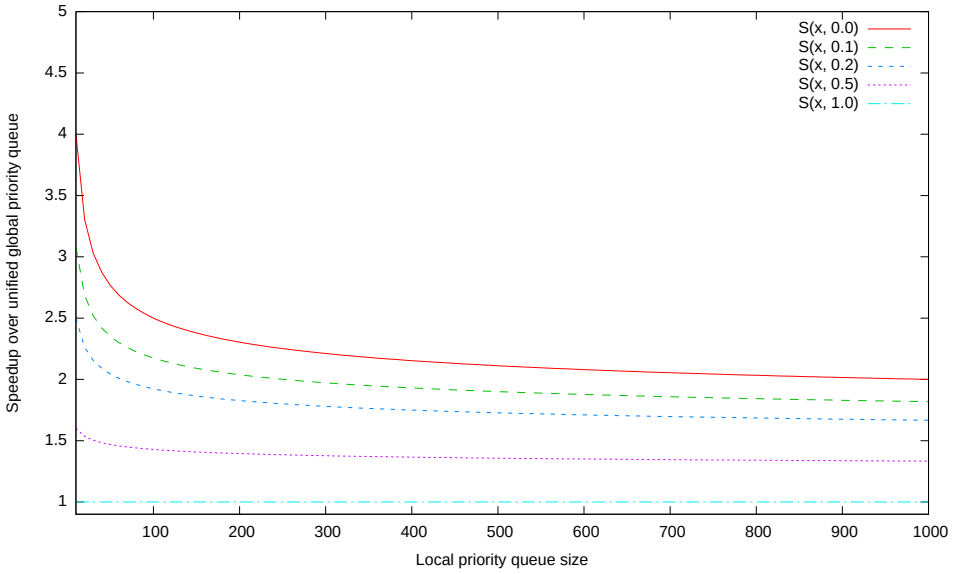

Figure 2.4: Speedup of decentralized over unified global priority queue

2.3 Neural Core

After having learned about how the scheduling in a neural net is performed, it's time to take a closer look at the base class NeuralEntity and the interface it describes. Each entity in a neural net – mainly neurons and synapses – inherits from class NeuralEntity. Figure 2.5 on page 21 gives a first impression about it's interface and the relation to other classes.

Each NeuralEntity has an *id* which uniquely identifies it within a neural net. You can access this id using the *get_id* and *set_id* methods. Once set, this shouldn't be modified as it is used as key in a Hash. Furthermore, as was mentioned already in the last section, each NeuralEntity contains a local priority queue – *stimuli_pq* – which stores all stimuli[4] that have not yet been processed as their timestamp lies ahead in the future.

To record specific events, for example the incidence of a neuron firing, every NeuralEntity includes a pointer to an instance of class *Recorder*. This can be accessed using the getter and setter methods *get_recorder* and *set_recorder*. If the pointer is `NULL` then no recording is performed for this specific instance of NeuralEntity.

Similarily, the Scheduler can be accessed using the methods *set_scheduler* and *get_scheduler*, inherited from class ScheduleEntity. Before a NeuralEntity can participate in a simulation, it must have a valid scheduler assigned. Also note that all entities must have the same scheduler assigned, otherwise scheduling will not work as expected.

2.3.1 Overview

To learn more about the purpose of the remaining methods *load*/*dump*, *connect*/*disconnect*, *stimulate* and *process* lets take a look at the steps involved in loading or creating a neural net, which are:

1. Allocating and initializing the entities of the net.
2. Connecting the entities according to the structure of the net.
3. Stimulating specific entities (initial stimulation).

Each logical distinct step depends on the former step being complete. For example you can't connect to an entitiy that does not yet exist, or, as stimulating an entity might send stimuli to connected entities, you should not stimulate an entity until it is fully connected.

When considering the simulation of a neural net, the first two steps shown above related to setting up a neural net are no longer appropriate. Instead the steps involved are reduced to the following two:

- Stimulating entities.
- Processing stimuli.

[4] Note that both words *stimuli* and *event* are used interchangably throughout the whole text.

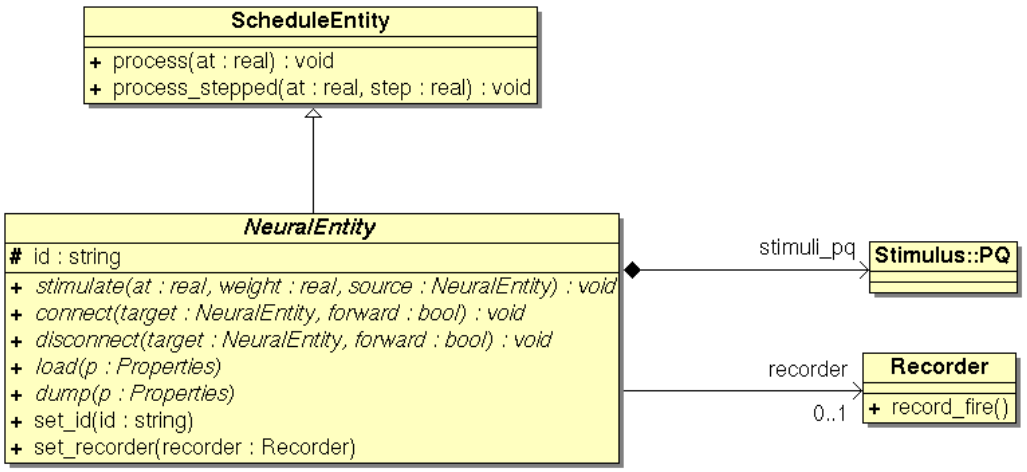

Figure 2.5: NeuralEntity and related classes

Stimulation of entities takes place for example when a neuron fires, in which case it stimulates its post-synapses, or a synapse which forwards a stimuli to its post-neuron. Processing of stimuli on the other hand means that in case of a neuron, it calculates its new state according to the incoming stimuli.

While stimulations either take place at present or lie ahead in the future, processing the stimuli always occur at present. The latter is required as we need to know all incoming stimuli in order to calculate the (exact) new state of a neuron, which we can't know until present unless we either do some form of prediction (optimistic scheduling) or are aware of the minimum propagation delay over all synapses; both would assume some global knowledge that we want to avoid in favor of a more generic simulation model.

Basically, you can also think of stimulation as a kind of communication that takes time to travel (propagation delay), whereas the processing is "doing the real work".

2.3.2 Allocating and initializing entities

Any concrete, i.e. non-abstract, subclass of *NeuralEntity* is allocated using the operator `new`. The constructor of any subclass should not take any arguments as this would prevent the constructors of all superclasses to be called automatically. This design-decision basically helps to reduce redundancy. Instead, methods should be favored over constructors with arguments to set the initial state to user-defined values.

Once an object of class *NeuralEntity* has been allocated, it should first be given an id and a scheduler instance assigned using methods *set_ id* and *set_ scheduler*. Afterwards, the internal state can be set using method *load*, which loads the internal state from the passed instance of class *Properties*. The latter basically serves as a dictionary containing key/value pairs. The whole procedure is further exemplified in the following listing:

```
// allocate
NeuralEntity *n = new Neuron_SRM01;

// initialize
n->set_id("n1");
n->set_scheduler(...);

// load internal state
Properties p;
p.set("abs_refr_duration", 0.5);
p.set("hebb", true);
n->load(&p);
```

The counterpart to method *load* is method *dump*, which writes the internal state of a *NeuralEntity* back into an instance of class *Properties*. While using a dictionary of key/value pairs (class *Properties*) for the purpose of setting or retrieving the state of an object might not be the most efficient approach, it is for sure the most generic one, and considering interfacing with foreign languages like Matlab or Ruby, this approach clearly pays off.

2.3.3 Connecting entities

Eminent for any net is that its building blocks are somehow connected to each other. The same applies, of course, to a neural net. In Yinspire, any *NeuralEntity* can be connected to any other *NeuralEntity* by using method *connect(target)*. It connects the receiver of the method (*this*) to the entity given as first parameter *target*. Imagine we want to connect entity A with B and B with C, we just write:

```
// NeuralEntity *A, *B, *C;
A->connect(B);
B->connect(C);
```

Again, this is a very generic interface to connect any two entities together. However, in a neural net consisting of neurons and synapses you can only connect a neuron with a synapse or a synapse with a neuron. Furthermore, to allow a specific method of learning in a neural net, the so called *Hebbsches learning*, an additional requirement for synapses and neurons is to know their pre-neurons and pre-synapses respectively, for the purpose of sending stimuli "the other way round". To meet this requirement, method *connect* takes a second parameter *forward* which describes the direction of the connection. This is used internally to "notify" the target that it has received a connection by calling it's *connect* method with the source as first parameter and the second parameter *forward* set to `false`. This should become obvious when looking

at the listing present in Figure 2.6 on page 25, showing the actual implementation of method connect for the classes Neuron and Synapse.

To remove a connection there is method *disconnect* with the exact same signature and notification protocol as used by method connect.

2.3.4 Stimulating entities

Another eminent fact about neural nets is that the components it is build up from – in general neurons and synapses – somehow communicate with each other by sending "signals". This communication can only take place alongside the path of the connections we talked about in the previous section. In other words, a neuron can send a signal to it's post-synapses, a synapse to it's post-neuron; in the case of *Hebbsches learning*, signals are also sent into the reverse direction. Most of the time those signals model a fire impulse originating either from a neuron or from an external stimulation source.

In Yinspire we call those signals *stimuli* or events. A stimulus (singular of stimuli) consists of a timestamp and a weight. To send a stimulus, the receiving entities' method *stimulate* is called. In case of an external stimulation, this would look like:

```
// NeuralEntity *n;
n->stimulate(4.5, Infinity, NULL);
```

Here we stimulate NeuralEntity n at time 4.5 with an infinitively strong stimulus (*weight*), without specifying the origin of the impulse (`NULL`). In Yinspire, a value of infinity has the meaning of *spontaneous firing*[5].

Another good example (see Figure 2.7 on page 26) for method *stimulate* can be found within class Synapse_Base, which implements the most basic model of a synapse by simply forwarding a stimulus adding a propagation delay. It does so by calling method *stimulate* of it's post-neuron with parameters `at + delay` and it's own weight attribute `this->weight`[6]. And to prevent infinite cycles in case of Hebbsches learning, the stimulus is only forwarded if it doesn't origin from the post-neuron itself.

In contrast to forwarding as performed by synapses, the default behaviour of a neuron is to store the received stimulus into its local priority queue and to reschedule itself if neccessary. See method stimulate of class *Neuron_Base* in Figure 2.7 on page 26 as an example, where the helper method *stimuli_add* of class NeuralEntity is used to store the stimulus into the local priority queue and to reschedule the NeuralEntity in the global priority queue if neccessary.

2.3.5 Processing stimuli

Back in Section 2.2 *Scheduling* we learned that method *process* is called by the Scheduler whenever a ScheduleEntity (and as such also a NeuralEntity) is scheduled for execution.

[5] This corresponds to an event type of `E_DO_FIRE` in Inspire.
[6] This is NOT the *weight* parameter passed-in from the neuron!

We also mentioned in another section that while stimulation usually involves the future, processing of stimuli is always performed at present.

So whenever the Scheduler calls method *process* (or *process_stepped*) the first parameter *at* refers to the current simulation time, i.e. the present. As at this point in time we know all the incoming stimuli that the NeuralEntity in question has received, we are able to calculate its new internal state according to the incoming stimuli. This is exactly what we do in the various models of neurons. Synapses on the other hand usually don't get scheduled because they just forward the stimuli and as such make no use of the local priority queue. However, new models of synapses, yet to be invented, might as well use that feature.

To better understand how processing of stimuli works in practice, see the pseudo-code given for method *process* in Figure 2.8 on page 26. It uses method *stimuli_sum* to sum up the weight of all stimuli in the local priority queue with timestamps less than *at*, while at the same time removing them from the local priority queue. Also note that method *stimuli_sum* will reschedule the NeuralEntity to the timestamp of the new top element (with timestamp $>$ *at*) of the local priority queue. As the result it returns the total weight of the input stimulus which is further used to calculate the new state (or potential) of the neuron. Once the potential exceeds a threshold, the neuron fires. Firing involves notifying the recorder that the neuron has fired, stimulating all post-synapses as well as resetting its own state.

2.3.6 A typical Interaction between Entities

A typical interaction between entities of different type (Neuron and Synapse) and the Scheduler is shown in the sequence diagram 2.9. Initially the Scheduler is under control (event loop) and schedules the next entity. In our case this is Neuron *n1*, whoose method process() with the current simulation time gets called. It is assumed that Neuron n1 has some local events that increase it's membrane potential such that the neuron fires. As such, it calls method fire(). Method fire() in turn stimulates all adjacent entities; in our case it calls method stimulate() of Synapse *s12*, passing the current simulation time as parameter *t* to tell the Synapse s12 that it fired at timestamp t. The implementation of method stimulate() of class Synapse is then to simply forward the stimulus but delaying it, i.e. the Synapse adds it's own propagation delay to the current simulation time and calls the stimulate method of it's post neuron *n2*. Now, the implementation of method stimulate() of class Neuron is different from that of a Synapse. Here we add the stimulus to the local priority queue using method stimuli_add(). This in turn might require that the entity has to be rescheduled in the global priority queue, which is done by calling the Scheduler's method schedule_update(). Once this returns, method stimulate() returns from Neuron n2, then stimulate() returns from Synapse s12 and we are back in method fire() of Neuron n1. If Neuron n1 would have more than one adjacent entity, it would call method stimulte() for the next adjacent entity and so on. After that the last adjacent entity has been informed that Neuron n1 has fired, method fire() returns after lowering it's membrane potential accordingly. Finally method process() returns from Neuron n1 and we are back in the main event loop of class Scheduler (schedule_run), where the next entity is scheduled and the same procedure starts over again.

```
void
Neuron::connect(NeuralEntity *target, bool forward=true)
{
  if (forward)
  {
    post_synapses.push(target);
    target->connect(this, false);
  }
  else
  {
    pre_synapses.push(target);
  }
}

void
Synapse::connect(NeuralEntity *target, bool forward=true)
{
  if (forward)
  {
    post_neuron = target;
    target->connect(this, false);
  }
  else
  {
    pre_neuron = target;
  }
}
```

Figure 2.6: Method *connect* of classes Neuron and Synapse

```
void
Synapse_Base::stimulate(real at, real weight, NeuralEntity *src
{
  if (src != post_neuron)
  {
    post_neuron->stimulate(at + delay, this->weight, this);
  }
}

void
Neuron_Base::stimulate(real at, real weight, NeuralEntity *src)
{
  stimuli_add(at, weight);
}
```

Figure 2.7: Method *stimulate* of classes Synapse_Base and Neuron_Base

```
void
process(real at)
{
  // sum (and remove) all stimuli with timestamps <= at.
  real weight = stimuli_sum(at);

  // calculate new state
  state = ....;

  if (state > threshold)
    fire(at, Infinity);
}

void fire(real at, real weight)
{
  if (recorder)
    recorder->record_fire(this, at, weight);

  stimulate_synapses(at, weight);

  state = 0.0; // reset state
}
```

Figure 2.8: Pseudo-code of method *process*

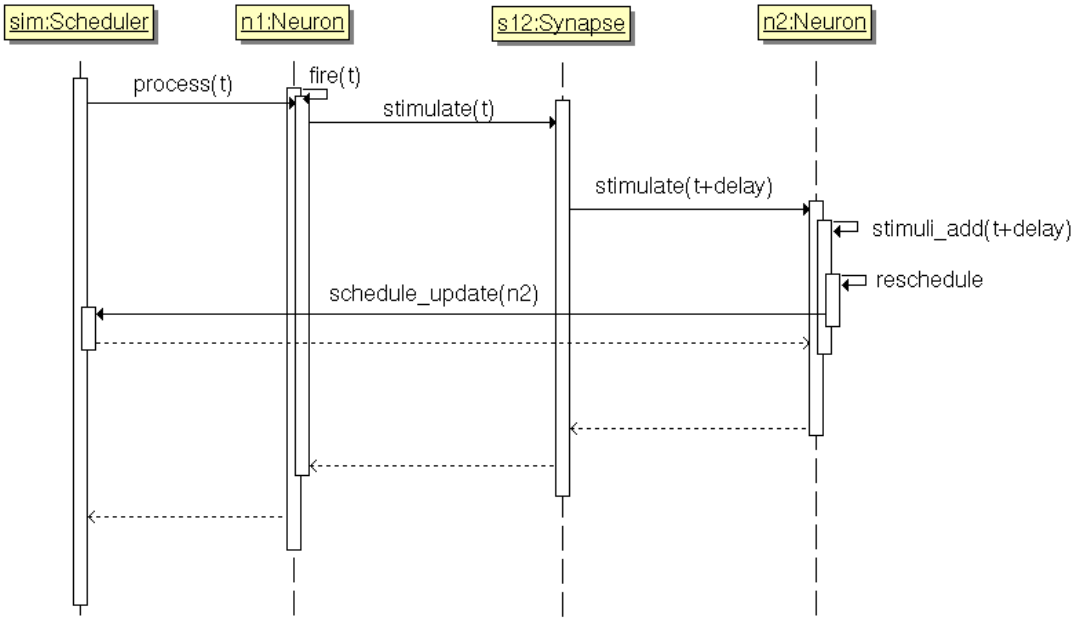

Figure 2.9: A typical interaction between entities of different types.

2.4 Structural Entities

In a neural net there are two structural distinct entities: Neurons and synapses. While neurons can have many pre- and post-synapses, synapses must have exactly one pre- and one post-neuron. The classes Neuron and Synapse in Yinspire (both direct subclasses of NeuralEntity) pay attention to exactly this circumstance by implementing just the structure eminent for neurons and synapses respectively.

Class Neuron stores the pre- and post-synapses in two arrays *pre_synapses* and *post_synapses*. Class Synapse stores the pre- and post-neuron in instance variables *pre_neuron* and *post_neuron*. As C++ is very bad at automatically resolving recursive dependencies between two classes, and doing it by hand is ugly as well, they all are of type NeuralEntity. That is, nothing prevents you from storing a synapse in *pre_neuron* or *post_neuron*, or neurons in *pre_synapses* and *post_synapses*, other than commonsense.

For the purpose of convenience, class Neuron additionally contains the two methods *stimulate_pre_synapses(at, weight)* and *stimulate_post_synapses* which iterate over all pre- and post-synapses calling their *stimulate* method with the given *at* and *weight* parameters.

2.5 Models

In this section the behaviour of all concrete models of neurons and synapses are described. For an overview please take again a look at the bottom of Figure 2.1 on page 12. To avoid unnecessary repetition, behaviour that is common for neurons and synapses is extracted into superclasses like Neuron_Base and Synapse_Base. This does not mean that your own class has to inherit from one of those two classes as you could also directly subclass from NeuralEntity.

When describing the models in the following sections, for the purpose of readability, we flatten the inheritance hierarchy and just describe the leaf classes including all attributes of their superclasses.

2.5.1 Neuron_SRM01

Class *Neuron_SRM01* is based on the Spike Response Model (SRM) using a dynamic threshold to model refractoriness and corresponds to the *KernelbasedLIF* model of Inspire. The core of the implementation is given in Figure 2.10 on page 32 and is further described below.

Behaviour

Incoming stimuli (method *stimulate*) are rejected if they are known to fall within the current absolute refractory period (if any). Otherwise, they are stored in the local priority queue and the neuron is potentially rescheduled. It still might be the case that a stimulus stored in the local priority queue falls within an absolute refractory period if its occurrence is farther ahead in time, as we can't predict future absolute refractory periods.

Once the neuron gets scheduled (method *process*), we first sum up (and remove) all stimuli that occured prior to the current time *at* storing the sum of weights into variable *weight* (line 16). Afterwards we determine whether we are still in the absolute refractory period or not (line 22), in which case there's nothing further to do for us. If not within the absolute refactory period, we continue with calculating the new state of the neuron – its mebrane potential – according to equations (2.1) and (2.2). This involves an exponential decay of the membrane potential and the weight of all incoming stimuli (variable *weight*).

$$decay = \exp(\frac{-(t_{at} - t_{lastSpikeTime})}{tau_m}) \tag{2.1}$$

$$mempot = weight + mempot * decay \tag{2.2}$$

As calculating the decay of the membrane potential depends on the last spike time, the last spike time has to be updated as well (line 30). Then we calculate the dynamic threshold (lines 33-34) according to equations (2.3) and (2.4).

$$decayThreshold = \exp(\frac{-(t_{at} - t_{lastFireTime} - t_{absRefrDuration})}{tau_{ref}}) \tag{2.3}$$

$$threshold = constThreshold + refWeight * decayThreshold \tag{2.4}$$

Again this involves an exponential decay, but this time the duration since the last fire time (minus the duration of the absolute refractory period) is used instead of the duration since the last spike time. The dynamic threshold is used to model *ref*ractoriness after a firing, which is where the prefix and suffix in *ref_weight* and *tau_ref* stems from.

Finally, if the membrane potential exceeds the dynamic threshold, the neuron fires by sending stimuli to its post-synapses (in case of hebbsch behaviour to its pre-synapses as well) and resetting its membrane potential back to zero (lines 37-42).

The initial value of *last_fire_time* and *last_spike_time* has been carefully choosen to be `-Infinity` which has the nice property of avoiding any conditionals in the calculations shown above in regard to the border case when no former fire or spike occured. To prove correct behaviour, lets see the outcome of the calculated variables for this border case:

$$lastFireTime = lastSpikeTime = -\infty$$

$$delta = \infty \geq 0$$

$$decay = decayThreshold = \exp(-\infty) = {}^{1}/_{\exp(\infty)} = {}^{1}/_{\infty} = 0$$

$$mempot = weight$$

$$threshold = constThreshold$$

This is exactly the expected behaviour.

Parameters

The parameters of a *Neuron_SRM01* are given in Table 2.1.

Parameter	Description	Default value
abs_refr_duration	Duration of absolute refractory period	`0.0`
last_spike_time	Last spike time	`-Infinity` (never)
last_fire_time	Last firing time	`-Infinity` (never)
hebb	Hebbsch behaviour	`false`
tau_m	Membrane time constant	`0.0`
tau_ref	Threshold time constant	`0.0`
ref_weight	Threshold weight factor	`0.0`
mem_pot	Membrane potential	`0.0`
const_threshold	Constant threshold	`0.0`

Table 2.1: Parameters of *Neuron_SRM01*

```
1  void
2  stimulate(real at, real weight)
3  {
4    // Enqueue stimuli to local priority queue unless in
5    // absolute refractory period
6    if (at >= last_fire_time + abs_refr_duration)
7    {
8      stimuli_add(at, weight);
9    }
10 }
11
12 void
13 process(real at)
14 {
15   // sum up (and remove) all stimuli weights <= at
16   real weight = stimuli_sum(at);
17
18   // time since end of last absolute refractory period
19   real delta = at - last_fire_time - abs_refr_duration;
20
21   // Return if still in absolute refractory period
22   if (delta < 0.0)
23     return;
24
25   // Calculate new membrane potential
26   real decay = exp(-(at - last_spike_time) / tau_m);
27   mem_pot = weight + mem_pot * decay;
28
29   // Update last spike time
30   last_spike_time = at;
31
32   // Calculate dynamic threshold
33   real threshold = const_threshold +
34                    ref_weight * exp(-delta/tau_ref);
35
36   // Fire if potential exceeds threshold
37   if (mem_pot >= threshold)
38   {
39     mem_pot = 0.0;
40     last_fire_time = at;
41     stimulate_synapses(at, Infinity);
42   }
43 }
```

Figure 2.10: Implementation of *Neuron_SRM01*

2.5.2 Neuron_SRM02

Class *Neuron_SRM02* is based on the Spike Response Model (SRM) using a dynamic reset to model refractoriness and corresponds to the *SpecialEKernel* model of Inspire. The core of the implementation is given in Figure 2.11 on page 35 and is further described below.

Behaviour

The behaviour comes very close to that of model *Neuron_SRM01* as described in 2.5.1, with the following differences:

- All incoming stimuli are stored in the local priority queue regardless whether in absolute refractory period or not.
- Refractoriness after firing is modeled using a dynamic reset potential instead of a dynamic threshold.
- Mebrane potential is calculated regardless whether in absolute refractory period or not.

The listing in Figure 2.11 on page 35 doesn't show method *stimulate*. This is because the default implementation inherited from class *Neuron_Base* is used, which is just `stimuli_add(at, weight)`, i.e. stimuli are stored regardless of absolute refractory period.

In method *process* the membrane potential is calculated in the same way as for *Neuron_SRM01* specified in equations (2.1) and (2.2), with the exception that it is calculated regardless of the absolute refractory period. After the new membrane potential is calculated and the last spike time is updated, we test whether we are currently in an absolute refractory period or not (lines 17-18). Notice the position of this piece of code vs. the position in the implementation of *Neuron_SRM01*. If not in absolute refractory period, we then continue with calculating the dynamic reset (line 21) according to equation (2.5).

$$dynamicReset = reset * \exp(\frac{-(t_{at} - t_{lastFireTime} - t_{absRefrDuration})}{tau_{ref}}) \tag{2.5}$$

Only if the membrane potential exceeds the constant threshold plus dynamic reset, the neuron fires. The actions taken when the neuron fires are somewhat more complex than those of a *Neuron_SRM01*. At first, we distinguish a spontaneous firing (a firing with an infinite weight) by testing the membrane potential for infinity (line 26), in which case we reset the membrane potential to zero and *reset* to *-u_reset* (lines 29-30). Otherwise we just add *u_reset* on top of dynamic reset and store it back to instance variable *reset* (line 34). Finally, we reschedule the entity for the end of its absolute refractory period (lines 38-40) and stimulate its post-synapses (line 43).

Parameters

The parameters of a *Neuron_SRM02* are given in Table 2.2.

Parameter	Description	Default value
abs_refr_duration	Duration of absolute refractory period	`0.0`
last_spike_time	Last spike time	`-Infinity` (never)
last_fire_time	Last firing time	`-Infinity` (never)
hebb	Hebsch behaviour	`false`
tau_m	Membrane time constant	`0.0`
tau_ref	Reset time constant	`0.0`
reset	Reset potential	`0.0`
u_reset	Reset offset	`0.0`
mem_pot	Membrane potential	`0.0`
const_threshold	Constant threshold	`0.0`

Table 2.2: Parameters of *Neuron_SRM02*

```
1  void process(real at)
2  {
3    // sum up (and remove) all stimuli weights <= at
4    real weight = stimuli_sum(at);
5
6    // time since end of last absolute refractory period
7    real delta = at - last_fire_time - abs_refr_duration;
8
9    // Calculate new membrane potential
10   real decay = exp(-(at - last_spike_time) / tau_m);
11   mem_pot = weight + mem_pot * decay;
12
13   // Update last spike time
14   last_spike_time = at;
15
16   // Return if still in absolute refractory period
17   if (delta < 0.0)
18     return;
19
20   // Calculate dynamic reset
21   real dynamic_reset = reset * exp(-delta/tau_ref);
22
23   // Fire if potential exceeds threshold
24   if (mem_pot >= const_threshold + dynamic_reset)
25   {
26     if (isinf(mem_pot))
27     {
28       // spontaneous firing (mem_pot == Infinity)
29       mem_pot = 0;
30       reset = -u_reset;
31     }
32     else
33     {
34       reset = dynamic_reset + u_reset;
35     }
36
37     // Schedule entity for end of absolute refractory period
38     if (abs_refr_duration > 0.0 &&
39         at + abs_refr_duration < schedule_at)
40       schedule(at + abs_refr_duration);
41
42     last_fire_time = at;
43     stimulate_synapses(at, Infinity);
44   }
45 }
```

Figure 2.11: Implementation of *Neuron_SRM02*

2.5.3 Neuron_LIF01

Class *Neuron_LIF01* is based on the Leaky Integrate and Fire (LIF) model and corresponds to the *ECurLIF* model of Inspire. It uses process-oriented simulation (stepped scheduling) to approximate the course of the membrane potential according to the input current.

Behaviour

The implementation for method *process* is given in Figure 2.12. Each incoming stimulus will increase or decrease (depending on the sign of the stimulus) the *input_current* (line 7). Then, stepped scheduling is enabled for this neuron to approximate the further course of the membrane potential (line 10).

Once stepped scheduling has been activated, method *process_stepped* (see Figure 2.13 on page 38) is called by the scheduler for each time step. Here we calculate the new membrane potential using Runge-Kutta to solve the integral. If the membrane potential exceed the constant threshold, the neuron fires. Stepped scheduling is deactivated if both the membrane potential and the input current fall below *mem_pot_bound* and *input_current_bound* respectively, until the next stimulus reactivates again stepped scheduling.

Parameters

The parameters of a *Neuron_LIF01* are given in Table 2.3.

Parameter	Description	Default value
abs_refr_duration	Duration of absolute refractory period	`0.0`
last_fire_time	Last firing time	`-Infinity` (never)
hebb	Hebsch behaviour	`false`
tau_m		`0.0`
tau_s		`0.0`
resistor		`0.0`
current_max		`0.0`
mem_pot	Membrane potential	`0.0`
mem_pot_bound	Membrane potential bound	`0.0`
input_current	Input current	`0.0`
input_current_bound	Input current bound	`0.0`
const_threshold	Constant threshold	`0.0`

Table 2.3: Parameters of *Neuron_LIF01*

```
1  void process(real at)
2  {
3    // Sum up (and remove) all stimuli weights <= at
4    real weight = stimuli_sum(at);
5
6    // Increase input current
7    input_current += weight * current_max / tau_s;
8
9    // Request stepped scheduling
10   schedule_enable_stepping();
11 }
```

Figure 2.12: Implementation of *Neuron_LIF01* (method *process*)

```
1  void process_stepped(real at, real step)
2  {
3    real current[3]; // incoming current at t, t+h/2, t+h
4    real factor = exp(-step / (2*tau_s));
5    current[0] = input_current;
6    input_current *= factor;
7    current[1] = input_current;
8    input_current *= factor;
9    current[2] = input_current;
10
11   if (at >= last_fire_time + abs_refr_duration)
12   {
13     // Calculate new membrane potential,
14     // integrate with Runge-Kutta
15     real k[4];
16     k[0] = step * f(current[0], mem_pot);
17     k[1] = step * f(current[1], mem_pot + k[0] / 2);
18     k[2] = step * f(current[1], mem_pot + k[1] / 2);
19     k[3] = step * f(current[2], mem_pot + k[2]);
20     mem_pot += 1.0/6.0 *
21       (k[0] + 2 * k[1] + 2 * k[2] + k[3]);// + O(step^5) error
22   }
23   else
24   {
25     mem_pot = 0.0;
26   }
27
28   if (mem_pot >= const_threshold)
29   {
30     // Fire
31     mem_pot = 0.0;
32     last_fire_time = at;
33     stimulate_synapses(at, Infinity);
34   }
35   else if (mem_pot < mem_pot_bound &&
36            current[0] < input_current_bound)
37   {
38     schedule_disable_stepping();
39   }
40 }
41
42 inline real f(real current, real u)
43 {
44   return ((-1.0 / tau_m * u) + (resistor / tau_m * current));
45 }
```

Figure 2.13: Implementation of *Neuron_LIF01* (method *process_stepped*)

2.5.4 Neuron_Input

To model an input neuron, there is class *Neuron_Input*. You don't need to use input neurons, as you can stimulate any neuron from an external source, not just input neurons. The difference is that an input neuron doesn't have any parameters (except *hebb*) and will always forward the stimuli it receives from it's pre-synapses to it's post-synapses, each at its appropriate time. For example if it receives three stimuli for times 1.0, 2.0 and 3.0, it will schedule itself for 1.0, 2.0, 3.0 and each time it gets scheduled it will forward the corresponding stimulus to it's post-synapses. As such an input neuron basically acts as a timed queue.

One interesting application of an input neuron is to act as a guard for neuron models that do not like infinite values of stimulation efficacy. Putting an input neuron in front of such a special neuron model either prevents misbehaviour or else reduces the special cases required for this special neuron model to be implemented. *Neuron_LIF01* for example is such a model that shows invalid behaviour in case it receives infinite stimuli (spontaneous fire requests).

The actual implementation is given in Figure 2.14. Method *stimulate* enqueues the received stimulus into the local priority queue (line 4), whereas method *process* iterates over each stimulus in the local priority queue (*stimuli_pq*) that occurs not later than *at* (line 10), removes it from the queue and stimulates all post-synapses (line 15). Afterwards it reschedules itself if neccessary according to the new top element of the local priority queue (lines 19-20).

```
1  void
2  stimulate(real at, real weight)
3  {
4    stimuli_add(at, weight);
5  }
6
7  void
8  process(real at)
9  {
10   while (!stimuli_pq.empty() && stimuli_pq.top().at <= at)
11   {
12     Stimulus s = stimuli_pq.top();
13     stimuli_pq.pop();
14
15     stimulate_synapses(s.at, s.weight);
16   }
17
18   // reschedule
19   if (!stimuli_pq.empty())
20     schedule(stimuli_pq.top().at);
21 }
```

Figure 2.14: Implementation of *Neuron_Input*

2.5.5 Neuron_Output

Class *Neuron_Output* is used to model output neurons. Similar to input neurons, an output neuron does not have any parameters and nothing prevents you from using any other neuron type for the purpose of recording fire times. The difference is that an output neuron will always record a fire for every stimuli it receives, so there is no calculation involved whether the neuron fires or not. The behaviour is mostly the same as that of an input neuron (see 2.5.4) with the exception that instead of forwarding the stimuli to the post-synapses it will just record the fire time (lines 15-16).

```
1  void
2  stimulate(real at, real weight)
3  {
4    stimuli_add(at, weight);
5  }
6
7  void
8  process(real at)
9  {
10   while (!stimuli_pq.empty() && stimuli_pq.top().at <= at)
11   {
12     Stimulus s = stimuli_pq.top();
13     stimuli_pq.pop();
14
15     if (recorder)
16       recorder->record_fire(this, s.at, s.weight);
17   }
18
19   // reschedule
20   if (!stimuli_pq.empty())
21     schedule(stimuli_pq.top().at);
22 }
```

Figure 2.15: Implementation of *Neuron_Output*

2.5.6 Synapse_Default

This is the most basic type of a synapse. It delays stimuli by *delay* and changes the forwarded stimulation efficacy to *weight*.

Behaviour

The core implementation of class *Synapse_Default* is given in Figure 2.16. Note that there is no implementation for method *process* as *Synapse_Default* neither makes use of scheduling nor of the local priority queue. Method *stimulate* simply forwards every incoming stimulus to its post neuron, given that it's not the post neuron itself that stimulates the synapse. The latter is required to prevent infinite cycles when using hebbsch behaviour, where stimuli are sent in both directions. The stimuli a *Synapse_Default* forwards is delayed for *delay* and is send with an efficacy of *weight* (the weight parameter of the synapse, not the method argument!).

Parameters

The parameters of a *Synapse_Default* are given in Table 2.4.

Parameter	Description	Default value
weight	Propagation efficacy	`0.0`
delay	Propagation delay	`0.0`

Table 2.4: Parameters of *Synapse_Default*

```
1 void
2 stimulate(real at, real weight, NeuralEntity *source)
3 {
4   if (source != post_neuron)
5   {
6     post_neuron->stimulate(at + delay, this->weight);
7   }
8 }
```

Figure 2.16: Implementation of *Synapse_Default*

2.5.7 Synapse_Hebb

Class *Synapse_Hebb* implements Hebbsch learning. It basically adapts its propagation efficacy (*weight*) according to whether forwarded stimuli led to a neuron firing or not. Note that in order to be useful you have to set the *hebb* property to `true` for all participating neurons; unlike Inspire there is no need for a special hebbsch neuron model.

Behaviour

The implementation of method *stimulate* is given in Figure 2.17. A *Synapse_Hebb* operates in two directions, the normal forward direction and the reverse direction from post-neuron to pre-neuron. In the forward direction (lines 6-23) it stores the timestamps of every incoming stimulus (*pre_synaptic_spikes*), adapts it's weight according to a learning algorithm, and finally forwards the stimulus like any regular synapse would do (line 23). In the reverse direction (lines 27-37), which results from the post-neuron signalling it's pre-synapse a firing, it adapts the *weight* as well according to a learning algorithm and finally resets the *pre_synaptic_spikes* array (line 37).

Parameters

The parameters of a *Synapse_Hebb* are given in Table 2.5.

Parameter	Description	Default value
last_fire_time		`-Infinity` (never)
current_fire_time		`-Infinity` (never)
weight	Propagation efficacy	`0.0`
delay	Propagation delay	`0.0`
learning_rate	Learning rate	`0.0`
decrease_rate	Decrease rate	`0.0`

Table 2.5: Parameters of *Synapse_Hebb*

```
1  void
2  stimulate(real at, real _weight, NeuralEntity *source)
3  {
4    if (source != post_neuron)
5    {
6      pre_synaptic_spikes.push(at);
7
8      if (last_fire_time > 0.0)
9      {
10       real delta_time = last_fire_time - at;
11       real delta_weight = learning_rate *
12         learning_window(delta_time);
13       uint sz = pre_synaptic_spikes.size();
14
15       if (sz > 1)
16         delta_time = pre_synaptic_spikes[sz - 2] - at;
17
18       delta_weight += decrease_rate * delta_time;
19       weight += (1.0 - fabs(weight)) * delta_weight;
20     }
21
22     // Forward stimulus
23     post_neuron->stimulate(at+delay, weight, this);
24   }
25   else
26   {
27     real delta_weight = 0.0;
28
29     last_fire_time = current_fire_time;
30     current_fire_time = at;
31
32     for (int i=0; i < pre_synaptic_spikes.size(); i++)
33       delta_weight += learning_rate *
34           learning_window(at - pre_synaptic_spikes[i]);
35
36     weight += (1.0 - fabs(weight)) * delta_weight;
37     pre_synaptic_spikes.clear();
38   }
39 }
40
41 inline real learning_window(real delta_x, real pos_ramp=1,
42   real neg_ramp=1, real pos_decay=10, real neg_decay=8) {
43   return ( (delta_x >= 0) ?
44            (pos_ramp * delta_x * exp(-delta_x/pos_decay)) :
45            (neg_ramp * delta_x * exp(delta_x/neg_decay)) ); }
```


Figure 2.17: Implementation of *Synapse_Hebb*

2.6 Miscellaneous Classes

There are some further classes in Yinspire worth to describe briefly. Please consult the well-documented source code for more in-depth information.

Recorder Used to record specific events like firing of a *NeuralEntity*. Every NeuralEntity stores a pointer to a *Recorder* instance.

NeuralNet A collection of *NeuralEntities*. Maintains an id to *NeuralEntity* mapping.

NeuralFactory Eases creation of model instances by type name, after registering the model class.

Simulator Glues together a *NeuralNet*, *Scheduler*, *NeuralFactory* and a *Loader_ Yi*n. Provides all functionality required to create, load and run simulations.

Loader Base class of all loaders. The purpose of a loader is to construct a neural net from a file.

Loader_Yin A loader for the Yin file format.

Dumper Base class of all dumpers. The purpose of a dumper is to dump a neural net back to a file.

Dumper_Yin Dumps a neural-net back to Yin format.

Dumper_Dot Dumps the structure of a net to the dot file format, which is part of graphviz, an application for laying out graphs.

```
Yin ::= (Template | Entity | Connect | Stimulate)*
Template ::= "TEMPLATE"? IdList "<" Type Parameters?
Entity ::= "ENTITY"? IdList "=" Type Parameters?
Connect ::= "CONNECT"? IdList ("->" IdList)+
Stimulate ::= "STIMULATE"? Id "!" Stimuli
IdList ::= Id ("," Id)*
Id ::= [A-Za-z_] [A-Za-z0-9_]*
Type ::= Id
Parameters ::= "{" Parameter* "}"
Parameter ::= Key "=" Value
Key ::= Id
Stimuli ::= "{" Stimulus* "}" | Stimulus
Stimulus ::= Weight "@" Timestamp | Timestamp
Value = Float | Bool
Timestamp ::= Float
Weight ::= Float
Bool ::= "true" | "false"
Float ::= ("+" | "-")? ("Infinity" | FloatNum)
FloatNum ::= Digits ("." Digits)? ("e" ("+" | "-")? Digits)?
Digits = [0-9] [0-9]*
```

Figure 2.18: EBNF of Yin File Format

2.7 The Yin File Format

Yinspire includes a loader (class *Loader_Yin*) for a human-readable file format called *Yin*, that provides an easy to learn yet flexible syntax to define neural nets. One of its features is to be able to define templates for commonly used neurons or synapses, which reduces typing a lot. Furthermore, input stimuli can be directly specified within this file format, making a separate file format superfluous.

The grammar of Yin in EBNF is given in Figure 2.18. Note that whitespaces are used to separate tokens where neccessary. Comments start with "#" and extend to the end of line. An example of a concrete Yin file is given in Figure 2.19.

```
TEMPLATE Neuron < Neuron_SRM01 {
  abs_refr_duration = 0.0
  const_threshold = 1.0
  ref_weight = 0.25
  tau_m = 1.15
  tau_ref = 3.0
}

# Excitatory synapse
TEMPLATE SynEx < Synapse_Default {
  delay = 0.7
  weight = 1.1
}

# Inhibitory synapse
TEMPLATE SynInh < Synapse_Default {
  delay = 0.2
  weight = -1.0
}

ENTITY n1, n2, n3 = Neuron

ENTITY e0, e1, e2 = SynEx
ENTITY i0, i1, i2 = SynInh

CONNECT n1 -> e0 -> n2 -> i0 -> n3
CONNECT n1 -> e1 -> n3
# ...

STIMULATE n1 ! 10.0
STIMULATE n2 ! {
  1.0 2.0 3.0 2@4.0
  5.0
}
```

Figure 2.19: Example Yin file

2.8 Converters

Yinspire includes converters for various file formats:

- *Netfile* – The format used by Gereon Weiss to describe neural-nets.
- *GraphML* – Neural-net definitions in GraphML (XML language for graphs).
- *SpikeTrains* – Simple format used to represent input stimuli.
- *Yin* – Novel file format introduced with Yinspire.

The *Netfile* format is very inflexible and hard to read/write for humans. But it is very fast and easy to parse. *GraphML* on the other hand is somewhat harder to parse. The implementation in Inspire is unusable for larger nets as it takes much longer to load a net than to simulate it. Editors exist to graphically generate nets in GraphML. Both, *Netfile* and *GraphML* formats require a separate format for spikes (*SpikeTrains*). The *Yin* format unifies neural-net definition and spike-definitions into one file format, is very fast to parse and human-readable.

The *tools* directory of the Yinspire distribution contains the following converters, all written in Ruby:

- GraphML_To_Netfile.rb
- GraphML_To_Yin.rb
- Netfile_To_Yin.rb
- SpikeTrains_To_Yin.rb

They all work by reading the to be converted file from standard input and print the result to standard output.

2.9 Command-Line Interface

The command-line usage of the Yinspire binary `yinspire` (see Chapter 4) is given in Figure 2.20. The basic usage is to pass any number of files in Yin format as arguments:

```
yinspire file1.yin file2.yin
```

Passing multiple Yin files can become handy as this allows you to modularize template declarations and to separate neural-net definitions from input stimuli defintitions.

By default no fire times are recorded unless you specify a file to record the fire times to, using the `--record` directive. To record to standard output, specify "-". To stop a simulation at a specific time, use the `--stop-at` option.

```
Usage: yinspire [options] file [file...]
 --stop-at N              Stop simulator at N (default: Infinity)
 --record FILE            Record fires to this file
 --dump FILE              Dump net after simulation
 --dump-dot FILE          Dump net after simulation in dot format
 --version                Show version
 --help                   Show this message
```

Figure 2.20: Command-Line Usage of yinspire

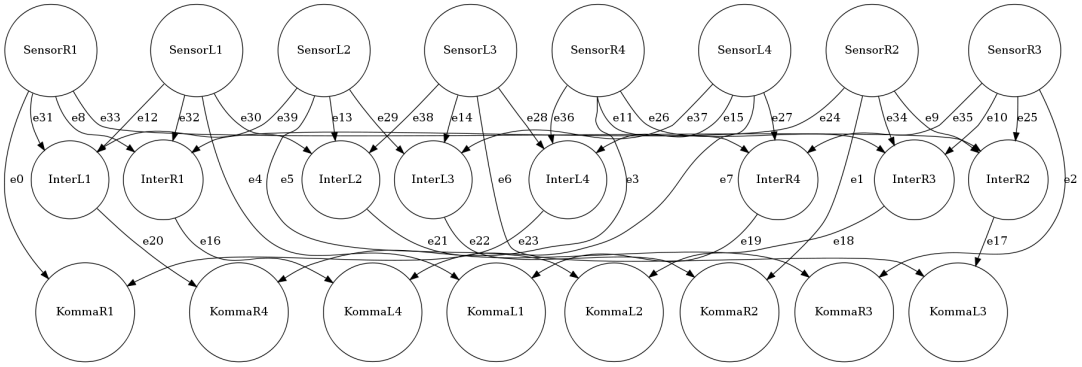

Figure 2.21: Graphical Representation of a Neural-Net using Graphviz/dot

The `--dump` option enables you to dump the whole neural-net after the simulation has ended to a file (in Yin format). Note that the dump does no longer contain any template definitions. For the purpose of visualization, the structure of the net can be dumped to the *dot* format of Graphviz[7] using the `--dump-dot` option. This can be used as follows, producing a graph as shown in Figure 2.21.

```
yinspire --stop-at -Infinity --dump-dot - skorpion.yin |
   dot -Tpng | display
```

[7] Graphviz http://www.graphviz.org/

3 Extending Yinspire

This chapter explains how to embed Yinspire into your application and how to extend Yinspire for new models of neurons or synapses.

3.1 Embedding Yinspire

At first, lets see how to embed Yinspire to use in your own C++ application. A basic example is given in Figure 3.1. The *Simulator.h* header file contains class *Simulator* and includes all necessary files. Class *Simulator* glues together a bunch of other classes (*NeuralFactory*, *NeuralNet*, *Scheduler*) and provides an easy-to-use interface for commonly used functionality. If you need the ultimate flexibility and functionality, you can still switch back using the underlying classes directly.

We then define our own recorder class *MyRecorder* (lines 5-14), which we use for entity "neuron1" (line 29). Method *record_fire* should be overwritten to provide custom behaviour. Any further details are described in the comments of the example.

3.1.1 Compilation

The next step is to compile your application. This requires that you have already compiled Yinspire as explained in Chapter 4. You now have two options: Either add your application to the *CMakeLists.txt* build script used by Yinspire, in the same way as is done for the command-line interface of Yinspire, or alternatively write your own Makefile. The latter is described in Figure 3.2 on page 53. The crux of the Makefile is to specify the include and library paths correctly, and to link against the *yinspirelib* library.

```
1  #include "Simulator.h"
2  using namespace Yinspire;
3  using namespace std;
4
5  class MyRecorder : public Recorder
6  {
7    public:
8
9      virtual void
10       record_fire(NeuralEntity *origin, real at, real weight)
11       {
12         // Do whatever you want here
13       }
14 };
15
16 int main(int argc, char **argv)
17 {
18   MyRecorder recorder;
19   Simulator sim;
20
21   // No recording by default
22   sim.set_default_recorder(NULL);
23
24   // Load nets
25   sim.load_yin("net.yin");
26   sim.load_yin("spikes.yin");
27
28   // Set recorder for entity "neuron1"
29   sim.get_entity("neuron1")->set_recorder(&recorder);
30
31   // Stimulate entity "neuron1"
32   sim.get_entity("neuron1")->stimulate(10.0, Infinity, NULL);
33
34   // Run simulation
35   sim.run(100.0);
36
37   // Dump net
38   sim.dump_yin("out.yin");
39 }
```

Figure 3.1: Embedding Yinspire in your Application (App.cc)

```
# CHANGE THIS!
YINSPIRE_ROOT=

INCLUDE_PATH=${YINSPIRE_ROOT}/src
LIBRARY_PATH=${YINSPIRE_ROOT}/build/Release

App: App.cc
	${CXX} -I${INCLUDE_PATH} -L${LIBRARY_PATH} \
		-lyinspirelib App.cc -o App
```

Figure 3.2: Makefile to compile App.cc

3.2 Implementing a new Model

Each model in Yinspire is implemented in a separate header file and resides within directory *src/Models*. Using header files is a lot easier because it reduces code redundancy, increases readability and doesn't require to modify the Makefile in case a new model is added, as it is explicitly included into another C++ file.

A template for a neuron model is given in Figure 3.3. This is a good start for your own model, all you have to do is to just replace every occurence of the word "Template" with the classifying name of your own model, e.g. *SRM03* or *LIF02*.

Lines 1-2 are required to prevent inclusion of this header file more than once. Make sure that you don't forget to modify those two lines according to the name of your model. Line 4 includes the class definition of our models' base class *Neuron_Base*. Note that all include paths are relative to the *src* directory of Yinspire.

The `DECLARE_ENTITY` macro in line 10 defines three methods (*create*, *ctype* and *type*), required to register the neuron model in a *NeuralFactory* and to allow the *NeuralFactory* to create an instance of your model.

Line 14 defines a property *my_property* for our model class. Here you would define all the properties that are required by your model. In the constructor in line 18, those properties get assigned a default value. Make sure to initialize any property in the default constructor and that you do not use any non-default constructors.

Loading and dumping of internal state is implemented in methods *load* and *dump*. Both call the superclass' method and then use macro `PROP_LOAD` (`PROP_DUMP`) to load (dump) values from (to) the Properties *p*. The macro is a shortcut for actually doing *p.load(my_property, "my_property")*, so by using this macro, the string is derived from the name of the property. If the externally visible name of the property, i.e. the name used in Yin files, should be different than the name of the instance variable, use the *p.load* method directly with the desired external name passed as second argument.

The actual neural-net related behaviour has to be implemented in the methods *stimulate* and *process* (and *process_stepped*) which are left empty in the template. An in-depth explanation about those issues are contained in Chapter 2.

Once your model is complete, the next step is to register the model with a *NeuralFactory*; class *Simulator* contains a *NeuralFactory* which can be accessed by calling method *get_factory*. For models shipped with Yinspire, this is performed in method *RegisterTypes* in file *src/RegisterTypes.h*. Registering a model is as simple as:

```
factory.register_type("Neuron_Template",
                      Neuron_Template::create);
// or
REGISTER_TYPE(factory, Neuron_Template);
```

The first argument to method *register_type* is the externally visible name for our model used in Yin files. It should equal the class name, in which case the `REGISTER_TYPE` macro should be favored.

```
1  #ifndef __YINSPIRE__NEURON_TEMPLATE__
2  #define __YINSPIRE__NEURON_TEMPLATE__
3
4  #include "Models/Neuron_Base.h"
5
6  namespace Yinspire {
7
8    class Neuron_Template : public Neuron_Base
9    {
10     DECLARE_ENTITY(Neuron_Template);
11
12     protected:
13
14       real my_property;
15
16     public:
17
18       Neuron_Template() : my_property(0.0) {}
19
20       virtual void
21         load(Properties &p)
22         {
23           Neuron_Base::load(p);
24           PROP_LOAD(p, my_property);
25         }
26
27       virtual void
28         dump(Properties &p)
29         {
30           Neuron_Base::dump(p);
31           PROP_DUMP(p, my_property);
32         }
33
34       virtual void
35         stimulate(real at, real weight, NeuralEntity *source)
36         {
37         }
38
39       virtual void
40         process(real at) {}
41   };
42 } /* namespace Yinspire */
43
44 #endif
```

Figure 3.3: Template for a Neuron model

```
// ...

#include "Models/Neuron_Template.h"

int main(int argc, char **argv)
{
  MyRecorder recorder;
  Simulator sim;

  REGISTER_TYPE(sim.get_factory(), Neuron_Template);

  // ...
}
```

Figure 3.4: Changes to App.cc to use our new model *Neuron_Template*.

The necessary pieces of code required to use our new model *Neuron_Template* in the application we developed in Section 3.1 are shown in Figure 3.4.

4 Installation

To compile Yinspire you need a C++ compiler and the cross-platform make tool CMake[1]. The latter was used to ease compilation especially on Windows systems.

4.0.1 Unix

To compile Yinspire on an unixoid system (Linux, *BSD, Solaris, MacOS X), issue the following instructions:

```
# Change into the root directory of Yinspire
cd yinspire

# Create build directory
mkdir -p build/Release

# Create makefiles
cd build/Release
cmake ../..

# And compile
make
```

This will create the binary `build/Release/yinspire` unless compilation fails. To compile a binary for the purpose of debugging, set the value of *CMAKE_BUILD_TYPE* to *Debug* when calling `cmake`. Or if you prefer to use double precision floats, set *YINSPIRE_DOUBLE_PRECISION* to *ON*. Both is shown in the example below:

```
cmake -DYINSPIRE_DOUBLE_PRECISION=ON \
      -DCMAKE_BUILD_TYPE=Debug ../..
```

4.0.2 Windows

On Windows, it is recommended to use the Microsoft Visual C++ Express Edition[2] compiler, even though compilation with MinGW should work[3] as well.

[1] CMake http://www.cmake.org/
[2] Only tested with version 8.0 (2005) and 9.0
[3] I've seen strange behaviour when compiled with MinGW and -O3 optimization.

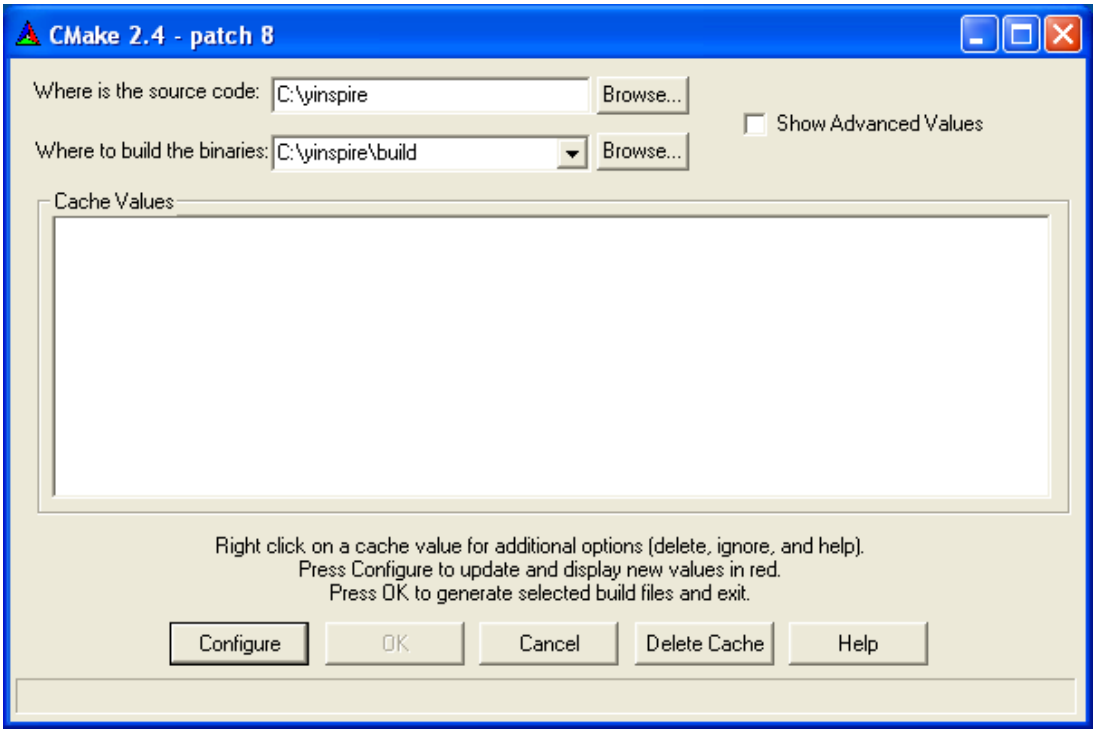

Figure 4.1: CMake on Windows

Start CMake and specify the paths to the source code and the build directory as shown in Figure 4.1, then press the CONFIGURE button. Next you have to choose the generator (Figure 4.2). Choose a generator for the compiler you want to use. This brings us to the dialog where you can set some compile options (Figure 4.3). For example to compile with double precision floats *YINSPIRE_DOUBLE_PRECISION* should be set to ON. Once done, press CONFIGURE again. Finally, click the OK button. This creates the makefiles necessary to build the project and exits CMake.

Now change into the *build* directory and double click on *YINSPIRE.sln*, assuming you're using Visual C++ Express Edition as compiler. Once the IDE has opened up, choose the configuration (Release or Debug build) and press F7 to build the project (Figure 4.4). Depending on the configuration you've choosen, you'll find the binary in subdirectory Release or Debug. You can then start the binary from the command prompt as shown in Figure 4.5.

4.0.3 Cross-Compilation

It is also possible to cross-compile Yinspire for the Windows platform on an unixoid system. To do so, you need to install the MinGW cross-compiler for your operating system. Then, when configuring with CMake, set *YINSPIRE_CROSSCOMPILE_MINGW* to *ON*. To test the binary, you can use the Windows Emulator Wine[4].

[4]Wine http://www.winehq.org/

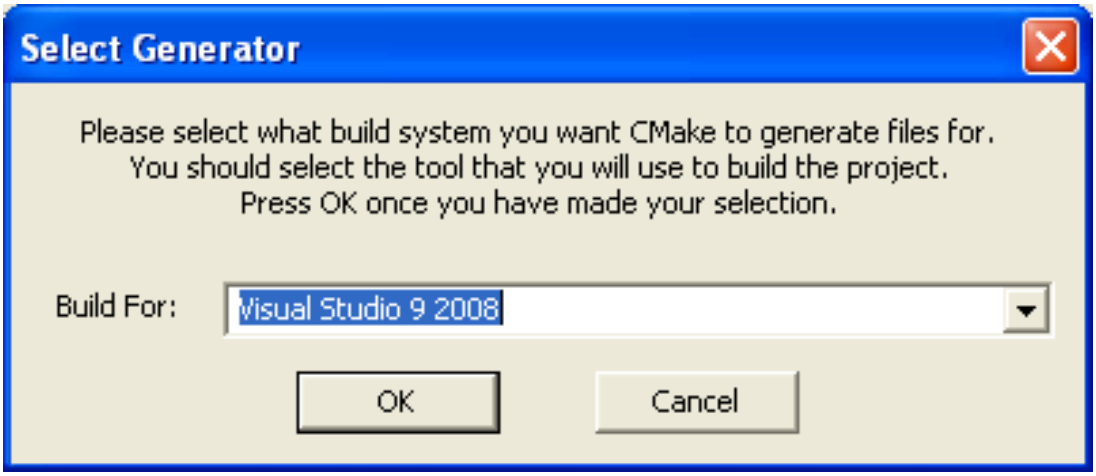

Figure 4.2: CMake - Choose the generator

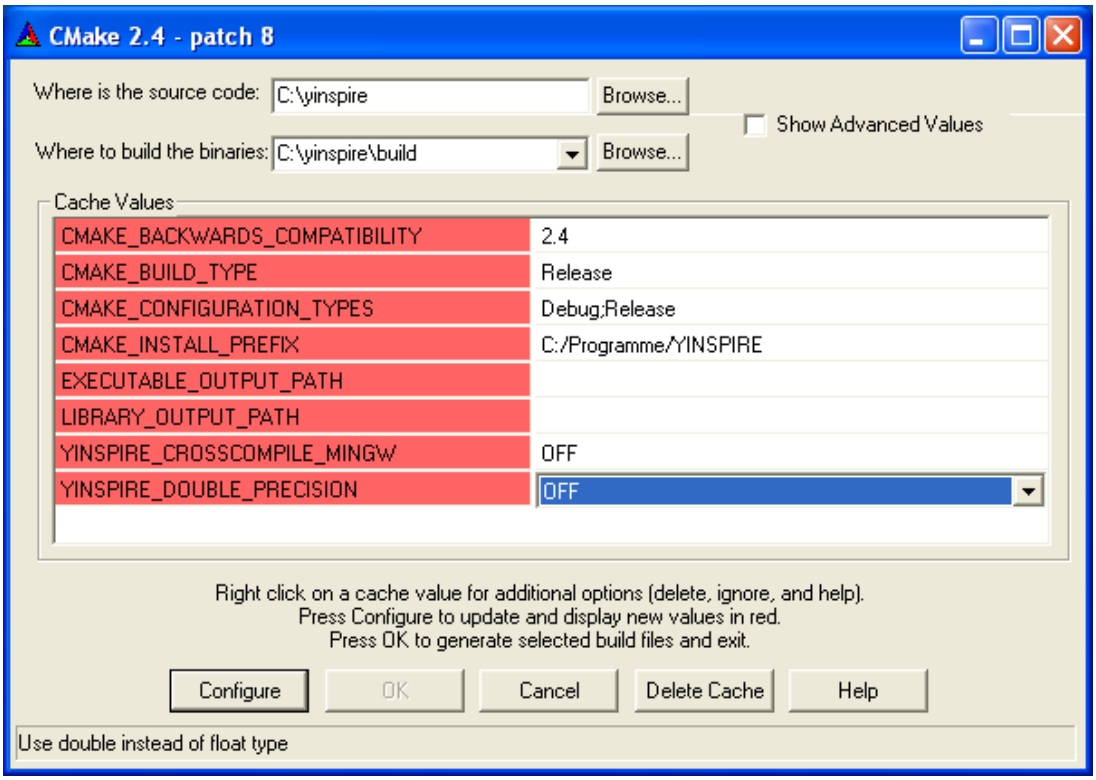

Figure 4.3: CMake - Set compile options

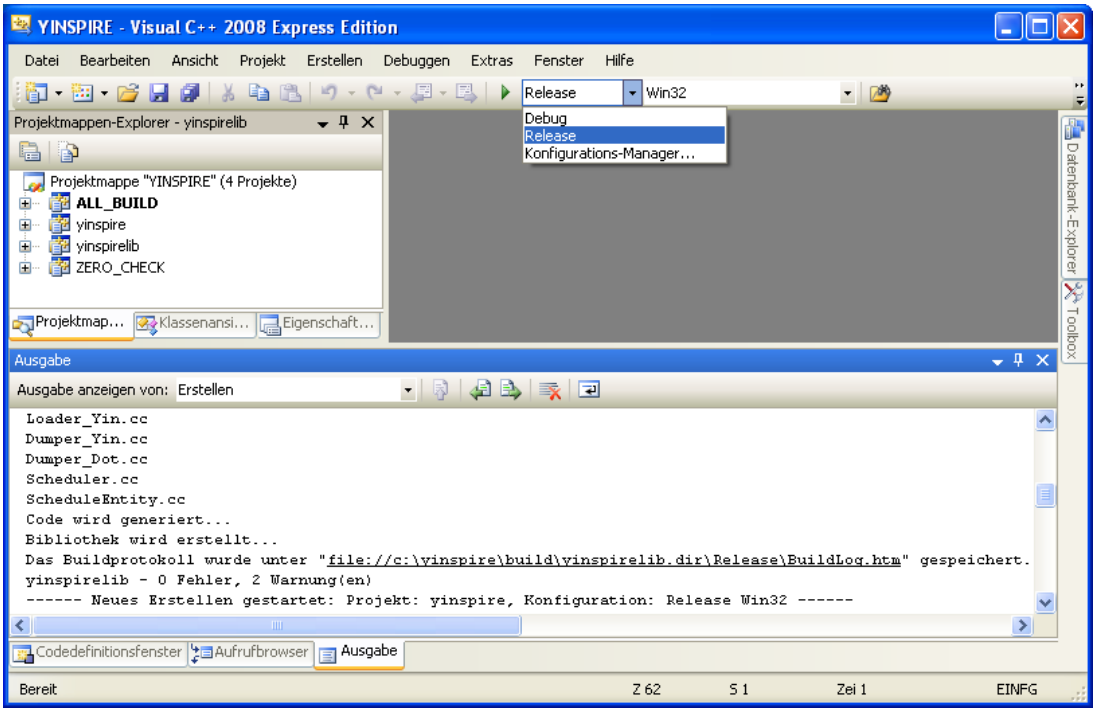

Figure 4.4: Building Yinspire with Visual C++ Express Edition

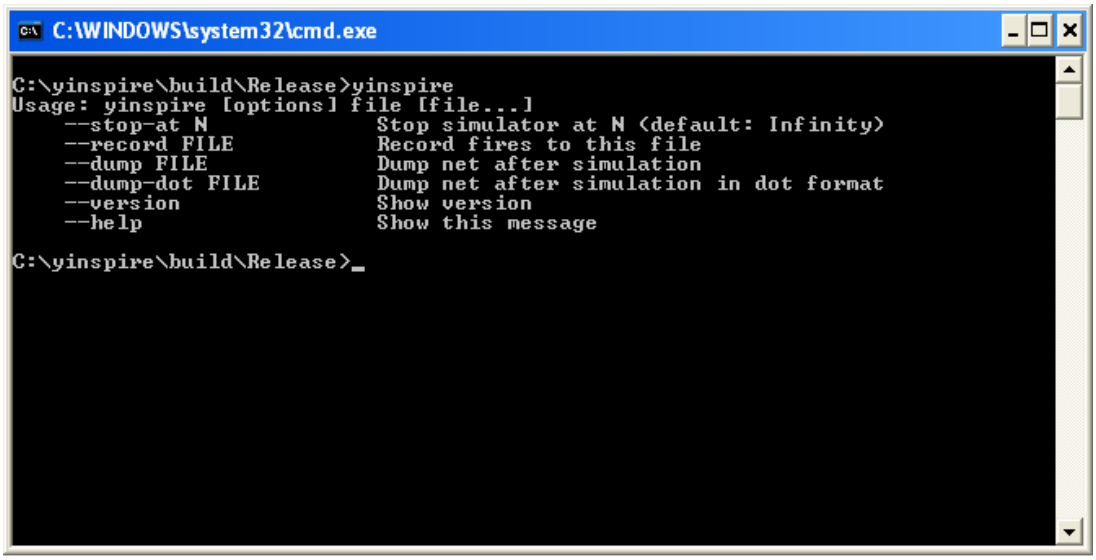

Figure 4.5: Running Yinspire from the Command Prompt

5 Interfaces to Foreign Languages

5.1 Ruby Interface

5.1.1 Installation

Only installation on Unixoid systems is described. Follow the steps in Chapter 4 to compile Yinspire. Then, assuming you have Ruby installed, issue the following commands to compile and install *ruby-yinspire*:

```
# Change into the directory of the Ruby interface
cd yinspire/ext/ruby-yinspire

# As root user, compile and install ruby-yinspire
su ruby setup.rb
```

5.1.2 Usage

The example in Figure 5.1 shows the basic usage of *ruby-yinspire*. All it does is to load two Yin files and then start the simulation.

A more complex example is given in Figure 5.2, demonstrating the usage of a custom *Recorder* as well as programmatically generating neural nets.

```
require 'yinspire'

Yinspire::Simulator.new do |sim|
  sim.load_yin('net.yin')
  sim.load_yin('another_net.yin')

  sim.run
end
```

Figure 5.1: Running a simulation with *ruby-yinspire*.

```
require 'yinspire'; include Yinspire

class MyRecorder < Recorder
  def record_fire(origin, at, weight)
    puts "#{ weight } ! #{ at }"
  end
end

Simulator.new do |sim|
  # Required to avoid the recorder instance from being
  # garbage collected.
  myrec = sim.register_recorder(MyRecorder.new)

  # Set default recorder
  sim.default_recorder = myrec

  # Create entities
  n1 = sim.create_entity 'Neuron_SRM01', 'n1'
  s1 = sim.create_entity 'Synapse_Default', 's1'

  # Connect them to a cycle
  n1.connect(s1)
  s1.connect(n1)

  # Set internal state
  n1.load :abs_refr_duration => 0.1, :tau_m => 1,
          :ref_weight => -0.05, :hebb => false
  s1.load :weight => 0.06, :delay => 0.5

  # Iterate over each entity in the net printing it's id
  sim.each_entity {|e| p e.id }

  # Access entity by id
  n1.id == sim['n1'].id

  # Stimulate n1
  n1.stimulate(0.0, Infinity)

  # Run simulation, stopping every 1000.0 ticks.
  while sim.run_for(1000.0)
    p sim.current_time
  end
end
```

Figure 5.2: Advanced example using *ruby-yinspire*.

5.1.3 Reference

The notation used below is the following: *Class.class_method* and *Class#method*. All classes are contained in the *Yinspire* namespace.

Simulator.new Creates new simulator instance.

Simulator.new{|sim|...} Creates new simulator instance and pass it to the block. At the end of the block, method *destroy* is automatically called.

Simulator#destroy Destroys simulator instance.

Simulator#load_yin(filename) Loads a Yin file from *filename*.

Simulator#dump_yin(filename) Dumps the net in Yin format to *filename*.

Simulator#current_time Returns the current simulation time.

Simulator#run(stop_at=Infinity) Runs the simulation until *stop_at*. Returns `true` if simulation hasn't yet reached the end (i.e. the priority queue is not yet empty). Otherwise `false`.

Simulator#run_for(ticks) Runs the simulation for *ticks* time steps (in simulator time). Returns `true` if simulation hasn't yet reached the end (i.e. the priority queue is not yet empty). Otherwise `false`.

Simulator#each_entity{|entity|...} Iterates over each *entity* in the net.

Simulator#get_entity(id) Returns a reference to the NeuralEntity named *id* in the net.

Simulator#[id] Alias for method *Simulator#get_entity*.

Simulator#create_entity(type,id) Creates a new entity instance of *type* and with *id*, and attaches it to the net.

Simulator#default_recorder= Sets the default recorder, which is used by every newly created NeuralEntity instance from now on.

Simulator#register_recorder(recorder) Instances of Recorder must be registered before being assigned to an NeuralEntity or used as default recorder. This is required to prevent them from being garbage collected, which could otherwise lead to dangling pointers.

Recorder.new Creates new recorder instance. Subclass, as the default behaviour is to do nothing!

Recorder#record_fire(origin,at,weight) Is called when NeuralEntity *origin* fires. Overwrite!

NeuralEntity#id Returns the NeuralEntity's *id*.

NeuralEntity#type Returns the NeuralEntity's model *type*.

NeuralEntity#dump Returns the internal state of the NeuralEntity as a Hash.

NeuralEntity#load(hash) Loads the internal state from *hash*.

NeuralEntity#connect(target) Connects itself to the NeuralEntity *target*.

NeuralEntity#disconnect(target) Disconnects itself from the NeuralEntity *target*.

NeuralEntity#stimulate(at,weight,source) Stimulates the NeuralEntity with the given parameters.

NeuralEntity#recorder Returns the used Recorder instance or `nil` if none such exists.

NeuralEntity#recorder= Assigns the Recorder instance to use for the particular entity (`nil` means "no recorder").

NeuralEntity#inspect Nicely outputs a NeuralEntity.

5.2 Octave/Matlab Interface

5.2.1 Installation

Only installation on Unixoid systems with Octave is described. Follow the steps in Chapter 4 to compile Yinspire. Then, assuming you have Octave installed, issue the following commands to compile *mex-yinspire*:

```
# Change into the directory of Octave/Matlab/Mex interface
cd yinspire/ext/mex-yinspire

# Compile Yinspire.mex
make compile

# Copy mex file to the other Matlab sources in directory m
cp Yinspire.mex m/
```

Once this has been accomplished, test whether it actually works by running Octave with a load path set to "m" as shown below:

```
# cd yinspire/ext/mex-yinspire
# octave -p m
octave-3.0.0:1> Simulator_new
lhs_1 = 705960352
ans = 705960352
octave-3.0.0:2>
```

```
sim = Simulator_new
Simulator_load_yin(sim, "net.yin")
Simulator_load_yin(sim, "another_net.yin")
Simulator_run(sim, 1000.0)
Simulator_destroy(sim)
```

Figure 5.3: Running a simulation with *mex-yinspire*.

5.2.2 Usage

The example in Figure 5.3 shows the basic usage of *mex-yinspire*. All it does is to load two Yin files and then running the simulation stopping at time 1000.0.

A more complex example is given in Figure 5.4, demonstrating how to programmatically generate neural nets.

5.2.3 Reference

Note that every function name is available both with and without a "Yinspire_" prefix.

Simulator_new() Creates new simulator instance.

Simulator_destroy(sim) Destroys simulator instance *sim*.

Simulator_load_yin(sim,filename) Loads a Yin file from *filename*.

Simulator_dump_yin(sim,filename) Dumps the net in Yin format to *filename*.

Simulator_run(sim,stop_at) Runs the simulation until *stop_at*. Returns the current simulation time (after the simulation).

Simulator_num_entities(sim) Returns the number of entities in the net.

Simulator_entity_ids(sim) Returns an array of all entity ids in the net.

Simulator_get_entity(sim,id) Returns a reference to the NeuralEntity named *id* in the net.

Simulator_create_entity(sim,type,id) Creates a new entity instance of *type* and with *id*, and attaches it to the net.

Simulator_set_default_recorder(sim,rec) Set the default recorder, which is used by every newly created NeuralEntity instance from now on.

Recorder_new Creates new recorder instance.

Recorder_destroy(rec) Destroys recorder instance *rec*.

Recorder_clear(rec) Clear all recorded fire times.

```
sim = Simulator_new

# Create recorder
rec = Recorder_new

# Set default recorder
Simulator_set_default_recorder(sim, rec)

# Create entities
n1 = Simulator_create_entity(sim, 'Neuron_SRM01', 'n1')
s1 = Simulator_create_entity(sim, 'Synapse_Default', 's1')

# Connect them
NeuralEntity_connect(n1, s1)
NeuralEntity_connect(s1, n1)

# Set internal state
x = {}
x.abs_refr_duration = 0.1
x.const_threshold = 0
x.ref_weight = -0.05
x.tau_m = 1
x.hebb = false
y = {}
y.weight = 0.06
y.delay = 0.5
NeuralEntity_load(n1, x)
NeuralEntity_load(s1, y)

# Stimulate n1
NeuralEntity_stimulate(n1, 0.0, Inf, 0)

# Run simulation
Simulator_run(sim, 1000.0)

# Get firetimes [(entity, at, weight)]
firetimes = Recorder_get_data(rec)

# Clear firetimes
Recorder_clear(rec)

Recorder_destroy(rec)
Simulator_destroy(sim)
```

Figure 5.4: Advanced example using *mex-yinspire*.

Recorder_get_data(rec) Return a cell array containing (entity, at, weight) tuples of all firetimes up to now.

NeuralEntity_id(ent) Returns the NeuralEntity's *id*.

NeuralEntity_type(ent) Returns the NeuralEntity's model *type*.

NeuralEntity_dump(ent) Returns the internal state of the NeuralEntity as a structured array.

NeuralEntity_load(ent,state) Loads the internal state from the structured array *state*.

NeuralEntity_connect(ent,target) Connects itself to the NeuralEntity *target*.

NeuralEntity_disconnect(ent,target) Disconnects itself from the NeuralEntity *target*.

NeuralEntity_stimulate(ent,at,weight,source) Stimulates the NeuralEntity with the given parameters.

NeuralEntity_set_recorder(ent,rec) Assign the Recorder instance to use for the particular entity.

6 Prototypes

6.1 Yinspire in Cplus2Ruby

During the process of implementing Yinspire in pure C++ I also made an attempt at creating a distinct version using a combination of Ruby and C++. The advantages of this version compared to the pure C++ version are manifold:

- No need to separate header (.h) and implementation files (.cc) as in C++. This improves readability and maintainability.
- No memory leaks due to (tracing) garbage-collection.
- Code that is not performance-critical can be written in Ruby, which is a lot more expressive and productive and less error-prone.
- Powerful scripting built-in. No need for arcane Matlab or Octave scripts.
- No need to manually write any wrapper code to interface a scripting language.
- Native C++ speed (for the parts written in C++).
- Meta-programming and code generation reduces code size further.

6.1.1 How it works

The by-product of this prototype is a library called Cplus2Ruby[1], which provides features to mix Ruby and C++ code in an object-oriented manner. To exemplify, see the example in Figure 6.1, which declares, in pure Ruby, a class *NeuralEntity* with a property *id* of type `Object` (any Ruby value can be assigned to it), and a subclass *Neuron* with a property *potential* of the C-type `float` and a method *stimulate*. The body of method *stimulate* (everything between `%{` and `}`) is pure C++ code. Note that Cplus2Ruby "abuses" Ruby as a Domain Specific Language (DSL), which is where Ruby really shines due to it's syntactical diversity. Even though the method body contains C++ code, the whole listing is 100% pure Ruby code. This is achived by using Ruby's String syntax `%{this is a string}` for the method body, and Hashes for the method signature.

At the time when line 14 (`Cplus2Ruby.commit`) is executed, Cplus2Ruby introspects the two classes and automatically generates the corresponding C++ code as well as code required to call the C++ methods directly from Ruby, which involves generating wrapper code that converts the arguments from C++ to Ruby types and vice versa.

[1] Cplus2Ruby http://rubyforge.org/projects/cplus2ruby/

```
1  class NeuralEntity; cplus2ruby
2    property :id
3  end
4
5  class Neuron < NeuralEntity
6    property :potential, :float
7
8    method :stimulate, {:at => :float}, {:weight => :float}, %{
9      potential += at * weight;
10   }
11 end
12
13 # generate code, compile and load library
14 Cplus2Ruby.commit("neural")
15
16 # use it
17 n = Neuron.new
18 n.id = "n1"
19 n.potential = 1.0
20 n.stimulate(1.0, 2.0)
21 print n.potential # => 3.0
```

Figure 6.1: Cplus2Ruby example code

```
1  class NeuralEntity : RubyObject {
2    VALUE id;
3
4    NeuralEntity() { id = Qnil; }
5  };
6
7  class Neuron : NeuralEntity {
8    float potential;
9
10   Neuron() { potential = 0.0; }
11
12   void stimulate(float at, float weight) {
13     potential += at * weight;
14   }
15 };
```

Figure 6.2: Approximate C++ code generated by Cplus2Ruby

It also automatically generates code required during the process of garbage collection. Once the code has been generated, a C++ compiler is fed with it to produce a shared library, which then is loaded. After the library has been successfully loaded, the C++ methods are available in Ruby. This all happens automagically in the background, without any user-interaction.

So what in line 17 happens, when we create a new object of class Neuron using `Neuron.new` is that, first of all, an instance of the Ruby class Neuron is created. Afterwards, an instance of the corresponding C++ class, the approximate source is given in Figure 6.2, is allocated and initialized. This C++ object is then attached to the Ruby object. To fully understand how this all works out one has to understand the internals of the Ruby interpreter, which is totally out of scope of this thesis. In short, Cplus2Ruby is a very powerful library to glue Ruby and C++ seamlessly together and there are a lot more features not described here.

6.1.2 Installation and Usage

This section only covers installation on Unix-like operating systems because Cplus2Ruby needs to invoke a C++ compiler at runtime which isn't by default available on Windows. Once you have Ruby[2] and RubyGems[3] (a package manager for Ruby) installed, getting Yinspire for Ruby installed is just a matter of typing `gem install yinspire` at the command prompt. Note that you might need to be the `root` user in order to perform this action. This will in turn download and install Yinspire for Ruby and all libraries it depends on. A command-line utility called `yinspire` is installed as well,

[2] Ruby Homepage http://www.ruby-lang.org/
[3] RubyGems http://www.rubygems.org/

```
Usage: yinspire [options]
 -s, --stop-at N            Stop simulation at N
                             (default: Infinity)
     --tolerance N          Stimuli tolerance (default: 0.0)
 -l, --load FILE[:FORMAT]   Load file
                             (formats: yin, spike, json, graphml)
 -d, --dump FILE[:FORMAT]   Dump net to file (after simulation)
                             (formats: yin, dot)
 -o, --output FILE          Filename to write output to
                             (default: stdout)
     --record-id x,y,z      NeuralEntity ids for which to record
                             fire events. (default: ALL)
     --record-type x,y,z    NeuralEntity types for which to record
                             fire events. (default: ALL)
     --do-not-simulate      Do not simulate
     --force-compilation    Force (re-)compilation of Yinspire
     --tmp DIR              Temporary directory for compilation
                             (default: /tmp/Yinspire)
 -h, --help                 Show this message
```

Figure 6.3: Command-Line usage of Yinspire/C++Ruby

which can be used according to Figure 6.3. For example `yinspire --load net.yin --record-id id1 --stop-at 1000` will load the Yin file *net.yin*, simulate it until the simulation time reaches *1000*, recording fire events for *id1* to standard output.

6.1.3 Conclusion

Even though this version has a lot of advantages, garbage collection to mention only one, it has been superseded by the pure C++ version. The main reason for this decision was that future students should still be able to extend this project. In the case of a mixture of Ruby and C++ this is much less guaranteed, especially when depending on a complex library like Cplus2Ruby, than for a version written in pure C++.

What this experiment has shown is the feasibility to implement a high-performance simulator for spiking neural nets in a garbage collected and dynamically typed language like Ruby, without loosing performance, given that the core is implemented in a lower-level language like C++.

6.2 An Editor for Neural Nets

To get an idea how a graphical user interface (GUI) for a neural net simulator could look like, a prototype was developed using the Ruby version of Yinspire, as introduced in Section 6.1. A screeshot is shown in Figure 6.4 below. For the GUI, the open-source,

cross-platform Fox-Toolkit[4] was used, which provides very good support for OpenGL and comes with a ready-to-use editor for three-dimensional objects.

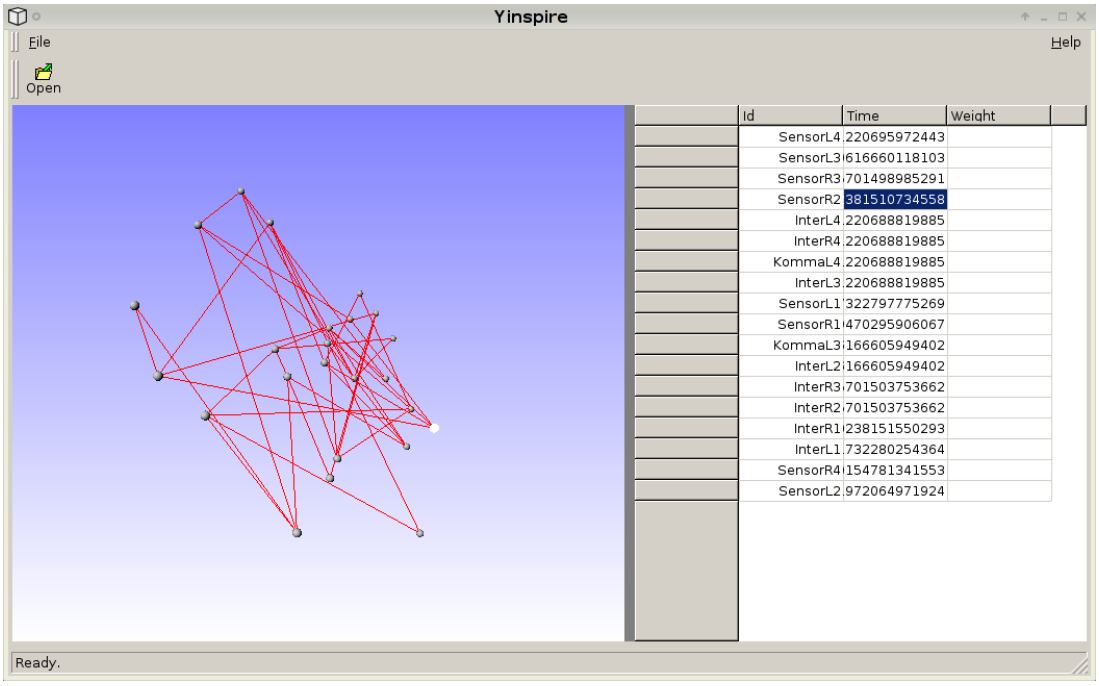

Figure 6.4: Screenshot of prototype Yinspire GUI

[4]Fox-Toolkit http://www.fox-toolkit.com/

7 Performance Benchmark

The primary research objectives for this thesis was to improve the performance of neural-net simulations relative to the existing simulator Inspire. To provide evidence that this goal has been successfully met, the underlying chapter describes a performance benchmark and presents the promising results.

7.1 Benchmark Setup

In the lack of a good, realistic benchmark, whose outcome could moreover serve as an indication for correct program behaviour, I have choosen the benchmark Gereon Weiss describes in Chapter 5 of his study thesis [Weiß, 2005].

He uses a total of 1000 neurons, of which 100 are input neurons, 700 are of type 1, and 200 of type 2. As Gereon neither provides the parameters he uses for the input neurons nor exactly describes whether input neurons can have incoming synapses or not, I make the assumption that input neurons have the same parameters as type 1 neurons and that they can have incoming connections. For the neuron model I've choosen *Neuron_SRM01*, which corresponds to *Neuron_KernelBasedLIF* in Inspire.

The parameters for all three types of neurons are given in Table 7.2. The synapse parameters – a function of the pre- and post-neuron type – is given is Table 7.4. The degree of connectivity is 10%, i.e. each neuron has 100 post synapses, leading to a total of 100,000 synapses. The stimuli given to the input neurons is 48 Hz over 10 seconds (which are 10,000 units of simulator time).

	tau_m	*ref_weight*	*tau_ref*	*abs_refr_duration*	*const_threshold*
input	20	-0.05	30	3	1
type 1	20	-0.05	30	3	1
type 2	30	-0.05	30	3	1

Table 7.2: Neuron parameters

7.2 Benchmark Procedure

Ten random nets were created with the structure as desribed in the Benchmark Setup section. Each net is simulated for 1, 10, 100, 1,000 and 10,000 simulator time units. The simulators used are *Inspire*, *Yinspire* and *Yinspire/double* – a variant of Yinspire which internally uses double precision floating point operations instead of single-precision

f	weight	delay
I/1 → I/1	0.06	1.5
I/1 → 2	0.09	0.8
2 → I/1	0.03	0.8
2 → 2	0.07	0.8

Table 7.4: Synapse parameters (I: input, 1: type 1, 2: type 2)

ones. The latter is used to measure the influence double precision has on performance[1], especially as Inspire uses double precision by default.

All three programs were compiled with `-O3 -DNDEBUG` flags using the Gnu Compiler Collection (gcc) in version 4.2.1. The machine used to run the benchmarks was the following:

- CPU: Intel Core 2 Duo (T7500) @ 2.2 GHz (dual core)
- Main memory: 1.5 GB
- OS: FreeBSD 7.0-STABLE i386

X11 and most other system daemons were disabled and no recording of fire times was performed. In a second step, the whole benchmark was repeated to measure scalability of the dual core CPU by running two simulator processes in parallel with the exact same simulations.

Note that the version of Inspire used is the one supplied with Gereon's study thesis. This step has become neccessary since the most recent versions of Inspire use a GraphML-based loader which itself takes far longer to load the net than to simulate it.

7.3 Results

Figure 7.1 shows the runtime averaged over all ten nets for the various simulators using a concurrency of 1 and 2. The increase in runtime for two simulations in parallel (concurrecy 2) is only marginally, which means that in this case the dual-core CPU (and the operating system) scales perfectly. The figure further illustrates that Inspire is slighly faster for simulation times less than 50. This is due to the more complex file format used for net and spike definitions in Yinspire which results in a higher startup cost. Apart from that, both Yinspire and Yinspire/double exhibit superior performance. This fact is again pictured in the speedup diagram in Figure 7.2, which shows for the 10-second case (simulation time 10,000) a 27-fold increase in performance for Yinspire over Inspire and still a 17-fold increase for Yinspire/double. For the full

[1] Because double precision requires twice as much memory as single precision and the most performance-critical operations occur inside the priority queue which copies a lot of memory around, it is assumed that this increase in memory copy operations will most likely be the reason for a decrease in performance.

data, including min and max runtimes, see Table 7.5. Memory usage is shown in Table 7.6.

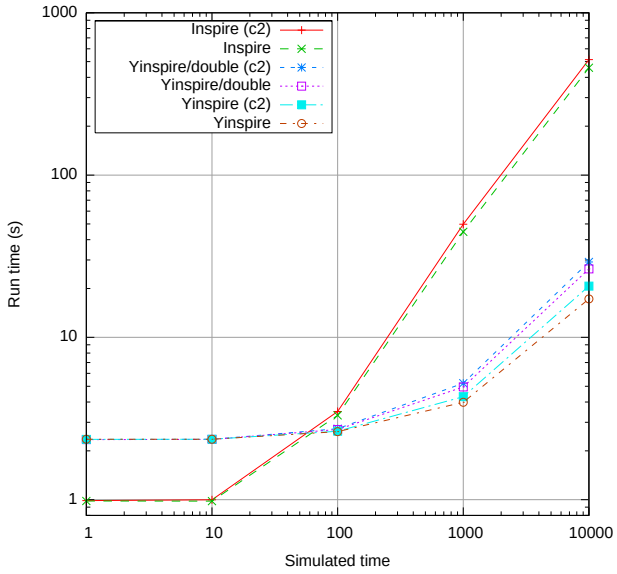

Figure 7.1: Runtime of Yinspire and Inspire

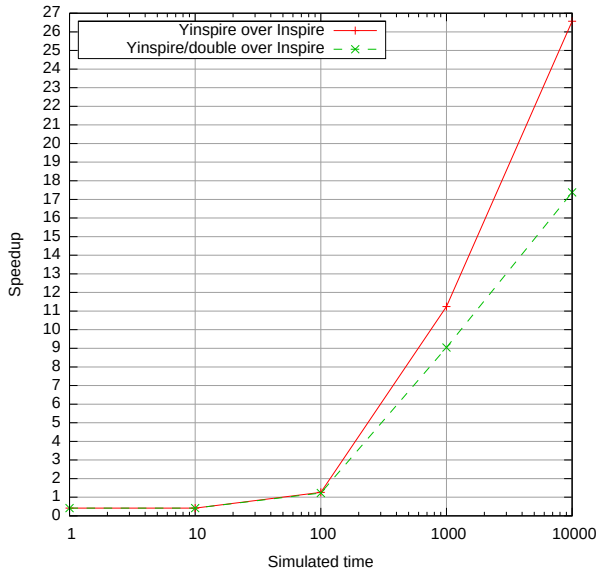

Figure 7.2: Speedup of Yinspire over Inspire

Simulated time	1	10	100	1000	10000
Concurrency 1					
Inspire T_{avg}	0.98	0.98	3.31	44.75	458.76
Inspire T_{min}	0.96	0.94	2.89	44.20	454.48
Inspire T_{max}	0.99	1.01	3.93	45.86	465.60
Yinspire T_{avg}	2.35	2.36	2.63	3.98	17.26
Yinspire T_{min}	2.33	2.34	2.56	3.87	17.10
Yinspire T_{max}	2.37	2.37	2.66	4.11	17.39
Yinspire/double T_{avg}	2.35	2.35	2.71	4.95	26.40
Yinspire/double T_{min}	2.33	2.34	2.63	4.73	25.91
Yinspire/double T_{max}	2.37	2.37	2.74	5.09	26.57
Concurrency 2					
Inspire T_{avg}	0.99	1.00	3.50	49.82	514.64
Inspire T_{min}	0.95	0.97	2.97	48.76	510.35
Inspire T_{max}	1.02	1.02	4.23	50.65	523.27
Yinspire T_{avg}	2.35	2.36	2.64	4.31	20.69
Yinspire T_{min}	2.33	2.34	2.56	4.16	20.01
Yinspire T_{max}	2.37	2.39	2.69	4.40	21.37
Yinspire/double T_{avg}	2.35	2.35	2.74	5.22	29.14
Yinspire/double T_{min}	2.33	2.32	2.66	4.96	28.12
Yinspire/double T_{max}	2.37	2.38	2.81	5.37	29.69

Table 7.5: Benchmark results. Runtime in seconds.

Simulated time	1	10	100	1000	10000
Inspire T_{avg}	20.03	20.04	20.04	20.04	20.04
Inspire T_{min}	19.97	20.00	20.00	20.00	20.00
Inspire T_{max}	20.05	20.05	20.05	20.05	20.05
Yinspire T_{avg}	15.61	15.68	16.52	16.52	16.54
Yinspire T_{min}	15.58	15.68	16.30	16.30	16.38
Yinspire T_{max}	15.63	15.69	16.59	16.59	16.59
Yinspire/double T_{avg}	17.74	17.89	19.40	19.41	19.43
Yinspire/double T_{min}	17.71	17.88	19.20	19.22	19.25
Yinspire/double T_{max}	17.78	17.90	19.50	19.50	19.50

Table 7.6: Benchmark results. Memory usage in MB.

7.4 Validation

To make sure that we don't compare apples and oranges in the benchmark shown above, all fire times were recorded separately for each simulation performed. To ease this task, Inspire was slightly enhanced (see Figure 7.3) to print out the fire times in the

same way Yinspire does. The number of fire times each simulator produces was then compared to assure that all three simulators (Inspire, Yinspire and Yinspire/double) do a similar amount of work, otherwise the numbers shown in Section 7.3 wouldn't be comparative at all.

The deviation of the number of fire times in relation to Inspire is given in Table 7.7. In the 10-second case (simulation time 10,000), the deviation is below 1.5% and can't influence performance much given the 17 to 27-fold increase in performance of Yinspire.

7.5 Conclusion

The benchmark present in this chapter clearly shows the superior performance of Yinspire, which outperforms Inspire in the 10-second case (also used by Gereon in his benchmark) by no less than 2700% (!) and still by 1700% when using double-precision floats.

```
--- a/KernelBasedLIFNeuron.cpp
+++ b/KernelBasedLIFNeuron.cpp
@@ -59,6 +59,10 @@ void KernelBasedLIFNeuron::fire(double now) {
         if (arpTime > 0) {
                 absRefPeriod = true;
         }
+
+#ifdef FIX_OUTPUT
+         cerr << id << "\t!\t" << now << endl;
+#endif
```

Figure 7.3: Patching Inspire to record fire times.

Net	1		10		100		1000		10000	
1	0.00	0.00	0.00	0.00	-3.07	-4.74	-0.16	-5.44	-0.01	-0.52
2	0.00	0.00	0.00	0.00	1.65	-2.50	0.21	-0.06	-0.42	-0.40
3	0.00	0.00	0.00	0.00	0.79	1.30	0.05	0.03	-0.86	-0.47
4	0.00	0.00	0.00	0.00	-0.44	-2.65	-4.85	-0.15	-0.92	-1.43
5	0.00	0.00	0.00	0.00	-0.09	-3.12	-4.60	-5.65	-0.83	-0.97
6	0.00	0.00	0.00	0.00	4.60	-0.08	-4.26	-5.22	-1.32	-0.93
7	0.00	0.00	0.00	0.00	7.67	2.73	0.62	-4.42	-0.76	-0.42
8	0.00	0.00	0.00	0.00	3.05	11.88	-3.38	-7.66	-1.15	-1.24
9	0.00	0.00	0.00	0.00	-1.29	0.93	-0.20	0.03	-0.46	-0.87
10	0.00	0.00	0.00	0.00	-5.38	0.40	-0.56	-5.30	-0.90	-1.35

Table 7.7: Deviation (in per cent) of the number of fire times of Yinspire (first subcolumn) and Yinspire/double (second) in relation to Inspire.

8 Benchmarking Priority Queues

As the most frequent and time-consuming operation in an event-driven simulator is to enqueue or dequeue an event into a priority queue, a great amount of time was spent to investigate various priority queue algorithms. This finally led to this benchmark, where the following algorithms were compared:

- (Implicit) Binary Heap
- Pairing Heap
- Calendar Queue
- StlPq – Priority Queue of STL[1] (uses Implicit Binary Heap as well)

The Classic Hold model [Jones, 1986] was used in the experiment with various distributions (Random, Exponential 1, Uniform 0 2, Triangular 0 1.5 and NegativeTriangular 0 1.5) and queue sizes from 2^{10} to 2^{20}. In the Classic Hold model, the size of the priority queue is constant and a so-called *hold* operation, which is a *dequeue* followed by an *enqueue* operation, is performed repetitive using the following procedure:

- *Setup*: Enqueue random elements until the priority queue reaches a size of n.
- *Warmup*: Perform w hold operations (10,000,000).
- *Overhead*: Measure the overhead of the loop and the pseudo random number generator by performing o empty hold operations (100,000,000).
- *Hold*: Perform m hold operations measuring the time it takes (100,000,000).

The value *enqueued* into the priority queue during each hold operation depends upon the value returned by the preceding *dequeue* operation according to:

$$enqueue(dequeue() + random)$$

with *random* being a random value conforming to one of the distributions mentioned above.

Three different element structures were used in the benchmark:

- *FLOAT* (4 bytes): `struct { float priority; }`
- *DOUBLE* (8 bytes): `struct { double priority; }`
- *STIMULI* (8 bytes): `struct { float priority; float weight; }`

[1] Standard Template Library of C++

8.1 Results

The following Figures 8.1 to 8.8 show the results of the benchmark, which was performed on the same machine as described in Section 7.2. All tests were performed twice with the results averaged. For the *Calendar Queue* and *Pairing Heap* algorithms only the *STIMULI* element structure was measured, and a chunked and free-list allocator was used to rule out potential slow memory allocation and to increase cache-friendliness.

Both my *Binary Heap* implementation and the *StlPq* algorithm exhibit similar performance, with *Binary Heap* being slightly faster. As the *DOUBLE* and *STIMULI* variants have to copy twice as much memory, their performance is slightly less than that of the *FLOAT* variant. Notice that the *Binary Heap/DOUBLE* results for small queue sizes are somewhat flawed, as they are actually faster than the corresponding FLOAT variant, which must be the result of an imprecise measure. The results of *StlPq/STIMULI* indicate that something must be going totally wrong, either in the compiler or in the implementation of the *StlPq* algorithm. It might be the case that a different algorithm is choosen depending on the element structure, but to strengthen this hypothesis a deeper analysis would become necessary.

The *Calendar Queue* algorithm exhibits for queue sizes up to 2^{17} near $O(1)$ runtime complexity and very good absolute performance. For larger queue sizes, performance drops down heavily, but still exhibits better performance than the *Pairing Heap* and better performance than all other tested algorithms for Random and NegativeTriangle distributinons. The *Calendar Queue* is also the algorithm whose runtime is most sensible to the distributions. The *Paring Heap* is in all disciplines slower than all other tested algorithms.

8.2 Conclusion

The most stable runtime behaviour with varying queue sizes and varying distributions exhibits the *Binary Heap* algorithm. It further provides good absolute performance, is well known and easy to implement. On the other hand, the *Calendar Queue* provides near $O(1)$ runtime complexity for queue sizes up to 2^{17} and within that range the best absolute performance of all tested algorithms. But it is hard to implement, not as general purpose as a Binary Heap as it requires monotonic increasing priorities, and runtime might be sensible to the value distribution. As such, the *Binary Heap* algorithm was used in Yinspire.

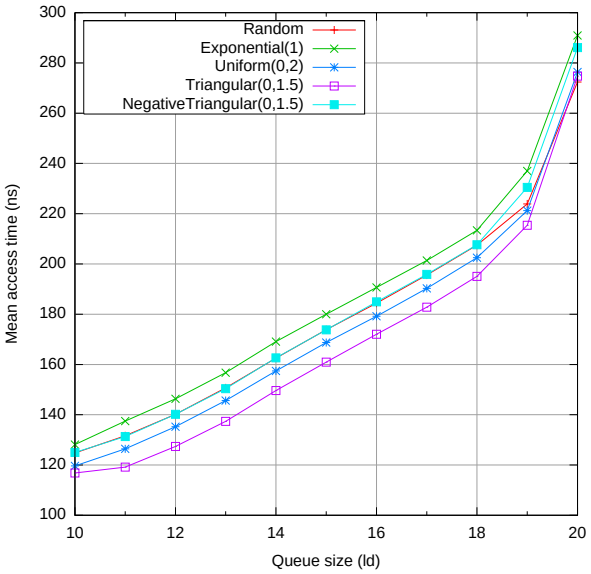

Figure 8.1: Binary Heap/FLOAT

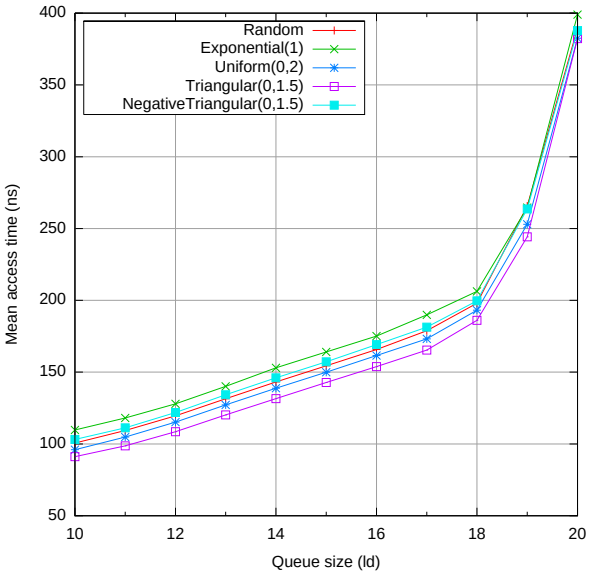

Figure 8.2: Binary Heap/DOUBLE

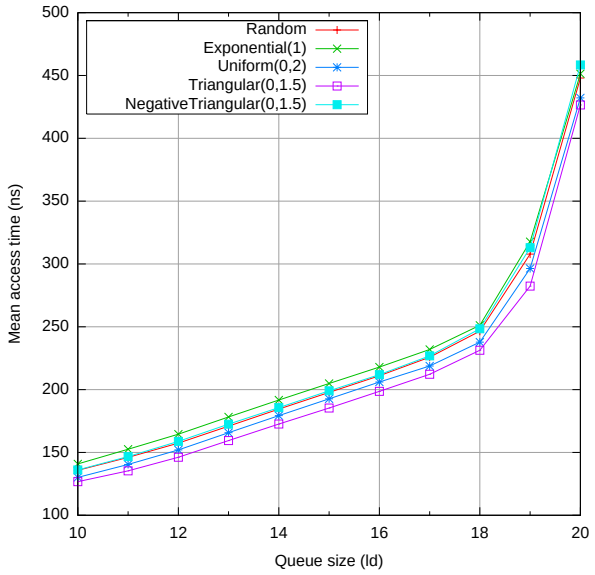

Figure 8.3: Binary Heap/STIMULI

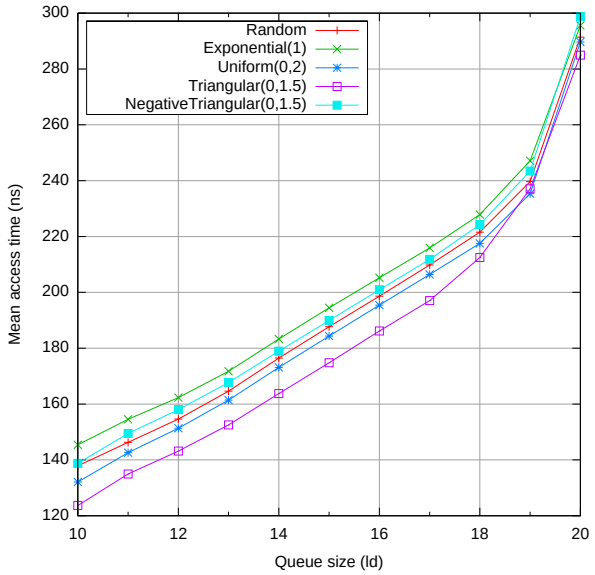

Figure 8.4: StlPq/FLOAT

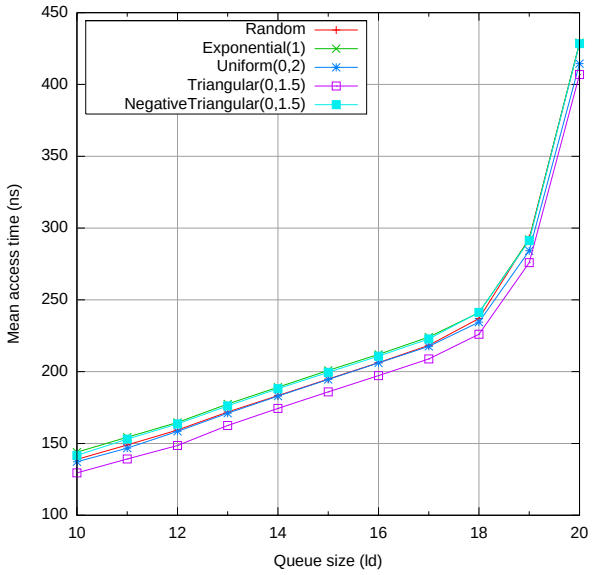

Figure 8.5: StlPq/DOUBLE

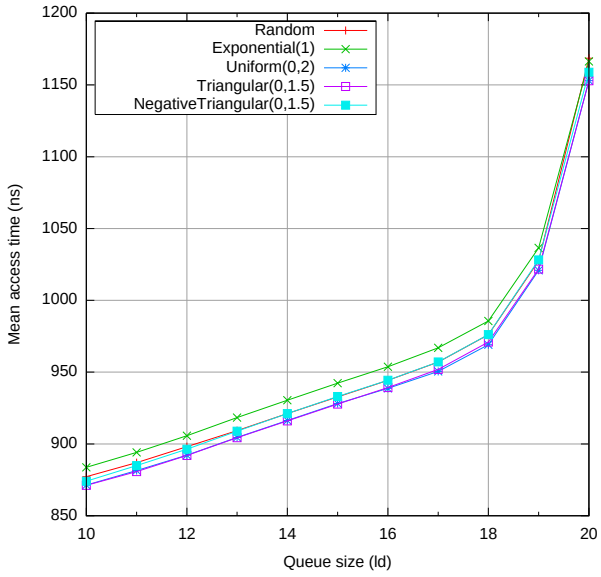

Figure 8.6: StlPq/STIMULI

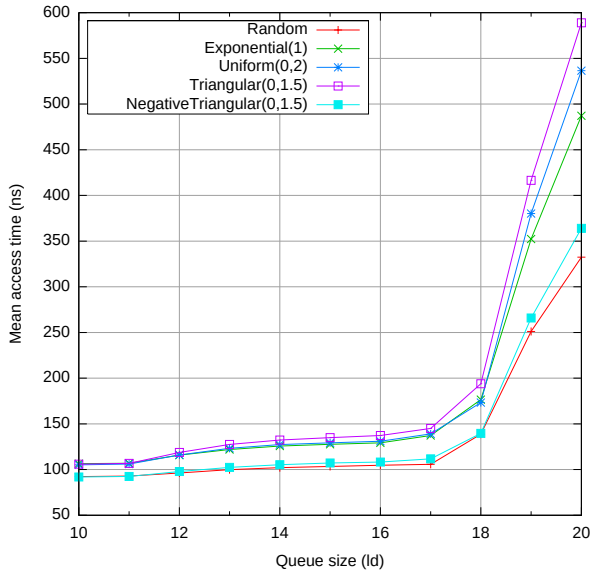

Figure 8.7: Calendar Queue/STIMULI

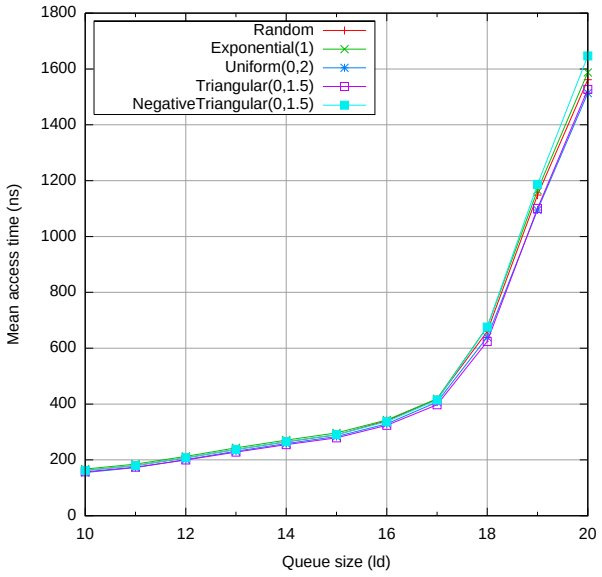

Figure 8.8: Pairing Heap/STIMULI

9 Performance Tips

This chapter describes a number of tips that can help to improve performance and code quality of an application. A lot of those tips have been applied in Yinspire and are responsible for it's extraordinary good performance.

While many approaches exist to improve the performance of an application, the first approach to consider should always be the choice of an adequate algorithm. But be careful, as too often it is the case that an algorithm with optimal (or good) time complexity performs worse on real hardware than an algorithm with non-optimal time complexity. One example are pointer-intensive tree algorithms that have showed good performance in the past but are supplanted nowadays more and more by index-oriented pointer-less (and cache-friendly) algorithms.

But there is more to performance than just the choice of an adequate algorithm as you can read below.

Keep it simple stupid

Complex software is hard to understand. Hard to understand software is even harder to optimize. Don't try to be too genius by using all the design patterns you've learned in the software engineering course if the only thing they add is complexity. Don't use more than 72 characters per line!

Avoid dynamic memory allocation

Dynamic allocation of memory can be quite slow on some systems[1]. Remember that dynamic memory allocation always involves some algorithms under the hood. The overall rule as such is to avoid dynamic memory allocation as much as possible, especially within time-critical routines. In case you need to allocate a lot of objects of the same type, investigate in using a free-list allocator together with chunked allocation.

Use generics (templates)

Algorithms that use templates in C++ can be *a lot* faster than their non-template counterparts due to improved compile-time specialization and optimization.

Use cache-friendly algorithms

Favor array-based storage over tree-algorithms that use pointers. Think about the penalties of a cache-miss.

[1] There is a reason why Firefox 3.0 ships with FreeBSD's memory allocator on Windows (and other) systems.

Avoid conditionals

Conditional branches can lead to processor-pipeline penalites due to mis-speculation. Avoid special cases by using sentinel values where possible. In Yinspire, this technique is used for example for the *last_fire_time* property which is set to negative infinity when no fire has yet occured. This avoids a flag *has_last_fire_time* and the corresponding conditional.

Avoid copying

In C++ and any other non-garbage-collected language, it is a hard problem to keep track about who is responsible for freeing dynamically allocated memory. One solution is to always copy the data avoiding references. This is for example the strategy used by *std::string* and in general a common technique found in C++. A garbage collector has the potential to lead to faster applications by avoiding the need to copy data.

10 Outlook

There is a lot left to be done (by others):

- A *web-interface* for Yinspire, where neural-nets can be uploaded and simulated on the server-side, without the need to install Yinspire locally. It would be further desireable to be able to batch-submit a lot of nets at once together with their simulation parameters, maybe using a web-service interface, and to get notified (via email) when the batch-jobs have completed.
- A *graphical editor* for neural-nets based on Yinspire, similar to the prototype shown in Section 6.2. Intelligent layout algorithms and real-time three-dimensional neural-net visualization would be advanced features.
- Investigate into: multi-list based priority queue algorithms like Ladder queue [Tang et al., 2005] or MList [Rick and Thng, 2003], cache-oblivious algorithms like Funnel Heap [Brodal and Fagerberg, 2002] and parallel priority queues [Rönngren and Ayani, 1997], [Sanders, 1998], [Grammatikakis and Liesche, 2000]. Benchmark each algorithm on current hardware.
- Design and implement new neuron and synapse models. For example multi-compartment model.
- Build a test-suite to ensure correct behaviour of Yinspire. This could go hand-in-hand with a benchmark-suite.
- Port Yinspire to Java, C# and/or D[1].
- Implement an embarrassingly parallel simulator for spiking neural net which runs on GPUs, using AMD Stream(tm)[2] and/or NVIDIA CUDA(tm)[3].

[1] D http://www.digitalmars.com/d/
[2] AMD Stream(tm) http://forums.amd.com/forum/categories.cfm?catid=328&zb=6888012
[3] NVIDIA CUDA(tm) http://www.nvidia.com/object/cuda_home.html

List of Figures

List of Tables

Bibliography

G. Brodal and R. Fagerberg. Funnel heap — a cache oblivious priority queue. In *Proc. 13th Annual International Symposium on Algorithms and Computation*, volume 2518 of *LNCS*, pages 219–228. Springer, 2002. URL citeseer.ist.psu.edu/brodal02funnel.html.

F. Feldbusch and F. Kaiser. Simulation of spiking neural nets with inspire me. eingereicht bei IEEE-SCM 05, 2005.

Miltos D. Grammatikakis and Stefan Liesche. Priority queues and sorting methods for parallel simulation. *Software Engineering*, 26(5):401–422, 2000. URL citeseer.ist.psu.edu/grammatikakis00priority.html.

D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications ACM*, 29:300–311, 1986.

Rick and Ian L. Thng. Mlist: an efficient pending event set structure for discrete event simulation. *International Journal of Simulation*, 4, December 2003.

Robert Rönngren and Rassul Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. Model. Comput. Simul.*, 7(2):157–209, 1997. ISSN 1049-3301. doi: http://doi.acm.org/10.1145/249204.249205.

Peter Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998. URL citeseer.ist.psu.edu/sanders97randomized.html.

Herbert D. Schwetman. Introduction to process-oriented simulation and csim (tutorial session). In *WSC' 90: Proceedings of the 22nd conference on Winter simulation*, pages 154–157, Piscataway, NJ, USA, 1990. IEEE Press. ISBN 0-911801-72-3.

Wai Teng Tang, Rick Siow Mong Goh, and Ian Li-Jin Thng. Ladder queue: An o(1) priority queue structure for large-scale discrete event simulation. *ACM Trans. Model. Comput. Simul.*, 15(3):175–204, 2005. ISSN 1049-3301. doi: http://doi.acm.org/10.1145/1103323.1103324.

Gereon Weiß. Ein geschwindigkeitsoptimierter simulator für gepulste neuronale netze. 2005.