“See” is a simulator for biologically detailed and artificial neural networks and systems.

- A model can be constructed from *simulation objects* each representing a population of neurons, a projection of axons, etc. These are connected by *data links* in a manner similar to MATLAB’s SIMULINK.

- A flexible and efficient protocol for data transfer between simulation objects ensures low overhead.

- The simulation environment can be used both *interactively* and in *batch mode*.

- The framework is organized around a Scheme interpreter providing a general purpose programming language which is used as model specification language, simulation control language, as well as implementation language.

- Time critical code and the basic framework for data flow is implemented in C++.

- The implementation is modular, using object orientation. This ensures easy extensibility.

- Several module libraries exist, e.g. an interface to the parallelized, platform independent library SPLIT intended for biologically detailed modeling of large-scale networks.
2 Structure

Guile  See is based on the embedded Scheme interpreter Guile from the GNU project. To achieve high efficiency, the low-level representation of model components, calculations and data flow is written in C++. Other code is written in Scheme. This resembles the architecture of the Emacs editor.

Scripting Language  Guile has been extended for neural modeling, giving a powerful language used for implementation, description of models, control of simulations and data analysis.

See modules  See has a modular architecture. Model components, e.g. an ANN population or a retina, are implemented by See modules with a standardized interface to the rest of the simulator.

Graphical User Interface  See has a graphical user interface which supports interactive construction of a model. Since it is written in Scheme, it is easily extensible by the user.

SPLIT  See contains an interface to the SPLIT library intended for biologically detailed modeling of large-scale networks. SPLIT can run on different types of parallel hardware, for example, clusters of workstations and vector computers. The asymptotic performance for large models is excellent.

SAROS  SIMON is an interface to simulators running in other processes, such as SAROS, a mechanical simulator.
During design of a general simulation package it isn’t possible to foresee all needs and requirements which might turn up during later use. When writing a general purpose package there is also the risk of introducing overhead due to considerations which are irrelevant given the needs of a specific user.

- One solution to the problem is to write a program, dedicated to simulating a specific model, in some general purpose language like C or C++.

- This means loosing the advantage of a general purpose package: that many technical problems encountered when building a model are already solved, e.g.:
  - Implementation of suitable data structures
  - How to organize data flow
  - How to choose and implement high quality numerical solvers
  - How to display data

- See is an alternative solution. While, on one hand, being a general purpose package, it is, on the other hand, easily extensible by the user in the same languages in which the simulator is implemented: C++ and Scheme.

- Example: See packages spike data in spike containers. Assume that such a container contains spikes recorded during an experiment. The user now wants to count the number of spikes between time from and time to. It is straightforward to device a tool for this task:
(define (count-spikes spikes cell from to)
  (let ((counter 0))
    (spikes-for-each (lambda (id time)
                       (if (and (>= time from)
                                 (< time to)
                                 (= id cell))
                          (set! counter (+ 1 counter)))
                      spikes)
     counter))

• The user now wants to calculate the average spiking frequency for a set of cells *cells* during the interval \([\text{from}, \text{to}]\). It is now easy to build on previous achievements:

(define (spike-average spikes cells from to)
  (let ((sum 0))
    (for-each (lambda (cell)
                (set! sum (+ sum (count-spikes spikes cell from to))))
              cells)
    (/ sum (length cells) (- to from))))

• It is evident that using a general purpose language as scripting language, i.e. for describing models and simulations, gives maximum flexibility.

• Apart from this, it can also be used to *extend* the simulator and *adapt* it to the needs of the user, as demonstrated above.
5 Modularity

See is built around the idea of interacting simulation objects, that exchange data through links. For example, a simulation object can be a population of neurons, a complex projection, a graphical display or data sources/recorders.

- Links connect the *interfaces* of two objects, not unlike cables and connectors on computers. Interfaces are low-level objects that handle data input or output. Several objects can connect to the same output interface.

- Links contain separate channels of data, not unlike the pins of a connector. For example, it might be desirable to send both spike and continuous data through the same link. The recipient objects can select which channels to use.

- Since all objects use the same framework for interconnection, it is possible to combine different types of objects in the same model.

- Internal state variables can be made available through *supervisor interfaces*. The supervisor interface enable user to read or change their values during the simulation.
• The complexity involved in moving data between components of a model is hidden by the interfaces. The programmer of a piece of dedicated simulation code only needs to declare an interface in order to create a source or sink for data.

• See is prepared for two means of parallelism:
  – Objects can be placed on different processors. Communication is handled by the interface code, and thus hidden from the programmer.
  – The objects themselves can be distributed over several processors. This is achieved through the use of parallelized standard vector libraries (e.g. the ANN module) or custom code (SPLIT).

• Beside the simulation objects, which participate in networks of data exchange, there are also objects which control the simulation, such as actors and sequencers.

• Actor objects perform pre-set actions at specific times, such as changing model parameters or the model itself.

• The simulation is event-driven, which means that events are ordered into a queue. When one event has been handled, the global time jumps to the value of the time-stamp of next event in the queue. Participating objects can have their own simulation timesteps; the simulator tells them when to simulate and to what point in the simulation to simulate.

• Sequencer objects implement this event based control. Sequencers can be embedded in sequencers, making it possible to use sub-simulations with different time resolution. It is also possible to synchronize the simulation to real-time. This is useful when it is important to gain an intuitive understanding of the dynamics.
When doing research it is important to be able to interactively experiment with models and throw together “sketches” for testing out new ideas.

- The environment needs to be as responsive as possible for interactive experimentation, with no need for long recompilations when changes are made. This is supported both through the interpreted scripting language and a graphical user interface to help set up models.

- Experimentation involves not just changing parameters in a fixed model, but also changing the model by adding or deleting simulation objects. In the modular framework of See it is possible to put together simulation objects like lego blocks, making it fairly easy to modify the model.

- A graphical interface has been written by Johannes Hjorth, enabling quick setup of models. It is written in Scheme.
- Setting parameters of the simulation object can be done interac-
tively in Scheme or using the graphical interface.

- Experiments and parameter searches can be defined in Scheme and
saved for the future. This enables testing different models with the
same experiment, and setting up scheme procedures to do extended
experiments overnight or longer.

- High performance simulation objects are implemented on the C++
level. It is also possible to create new simulation objects on the
Scheme level. It is possible to prototype new ideas in Scheme
alone, and if they are useful they can later be implemented in a
more efficient manner.
(let ((r (make-retina :params retinaparams))
    (m (make-multiproj))
    (c (make-cortex))
    (w1 (make-spike-window 1024 32))
    (w2 (make-spike-window 1024 32))
    (rc (make-recorder)))
  (connect! r m)
  (connect! m c)
  (connect! r 'on w1)
  (connect! r 'off w2)
  (connect! c 'e 'cont rc)
  (model! r m c w1 w2 rc))
10 Adding New Code

New Modules

- Writing new scheme routines is simple and can be done interactively. In addition, it should be simple to add dedicated C++ modules for interfaces with other software or high performance tasks.

- The procedure for adding a new module to See is
  1. Write the basic code.
  2. Add Scheme level interfaces. Modules can “publish” names in Scheme space, making it possible to access parameters, simulation objects or data from Scheme.
  3. Add the module to the See build tree and run Gnu configure to configure the system.
  4. Build the system.

The module becomes accessible on the Scheme level, and from now on declared simulation object types from the module can be used in models.

Network Objects

Simulation objects that can participate in a data exchange network, network objects, are ordinary C++ objects that inherit from the predefined class NetOb and implements a simple protocol. The network objects have one or more interface objects enabling data exchange and implements the following functions:

initialize() called when the object is created.

begin_simulation() called when the simulation is started.

simulate(Real t) Simulate up to time t.

close_simulation() - called when the simulation ends.
Yet another simulator?

Today, there exists several general purpose simulation packages for neural nets, for example Genesis, NSL, Neuron, Nodus and Swim. So, why yet another simulator?

- See was built to fill needs which cannot be fulfilled \textit{in combination} by other currently available packages. Of these, the following are the most important:

  \textbf{Systems of networks} It should be easy to compose \textit{systems} of networks from previously developed and tested models. It should be easy to handle \textit{large} networks.

  \textbf{Parallelism} We want to be able to utilize parallel hardware efficiently. This is a necessary requirement in order to handle large networks.

  \textbf{Polytypic models} Models may be built of components of different type. We may combine a biologically detailed network model, networks with ANN units and a mechanical model in the same simulation.

  \textbf{Modularity} It should also be easy to add dedicated code and interface this code to other parts of a simulation.

  \textbf{Embedded extension language} The user can easily adapt the simulator to his own needs through the use of the general purpose interpreted extension language.

- We believe that See can contribute some useful ideas to the development of the third generation of simulators.
Future Development

See is still under heavy development. Until now, the main focus has been to get a flexible and extensible infrastructure and to support “in-house” projects. Its use still requires thorough computer skills.

- Areas under current development are:
  - An object oriented specification language for easy specification of model parameters when working with networks or cells with complex internal structure. It will be based on the idea of multiple inheritance of parameter blocks utilized in the Swim simulator.
  - Further development of the Graphical User Interface, e.g. a graphical editor for the object oriented specification language.
  - Improvements of the scripting language to achieve more power and greater simplicity.
  - Work on parallelization code.

- While See is mainly intended for “in-house” use, we intend to make it available under the GNU General Public License.

Contact Information

- Mikael Djurfeldt djurfeldt@nada.kth.se
- Anders Sandberg asa@nada.kth.se

SANS
Department of Numerical Analysis and Computing Science
Royal Institute of Technology
S-100 44 Stockholm
Sweden
Tel: +46 (8) 790 69 09
Fax: +46 (8) 790 09 30