The state of MIIND

Marc de Kamps [1]

Biosystems Group, School of Computing, University of Leeds, LS29JT Leeds, United Kingdom

Volker Baier [2]

Neuro-Cognitive Psychology, Ludwig-Maximilians Universität München, Leopoldstrasse 13, München, Germany

Johannes Drever

Robotics and Embedded Systems, Institut für Informatik, Technische Universität München, Boltzmannstrasse 3, D-85748 Garching bei München, Germany

Melanie Dietz

Robotics and Embedded Systems, Institut für Informatik, Technische Universität München, Boltzmannstrasse 3, D-85748 Garching bei München, Germany

Lorenz Mösenlechner

Image Understanding & Knowledge-Based Systems, Institut für Informatik, Technische Universität München, Boltzmannstrasse 3, D-85748 Garching bei München, Germany

Frank van der Velde

Leiden Institute for Brain and Cognition, Cognitive Psychology, Leiden University, Wassenaarseweg 52 2333 AK Leiden, The Netherlands

Abstract

MIIND (Multiple Interacting Instantiations of Neural Dynamics) is a highly modular multi-level C++ framework, that aims to shorten the development time for models in Cognitive Neuroscience (CNS). It offers reusable code modules (libraries of classes and functions) aimed at solving problems that occur repeatedly in modelling, but tries not to impose a specific modelling philosophy or methodology. At the lowest level, it offers support for the implementation of sparse networks. For example, the library SparseImplementationLib supports sparse random networks and the library LayerMappingLib can be used for sparse regular networks of filter-like operators. The library DynamicLib, which builds on top of the library SparseImplementationLib, offers a generic framework for simulating network processes. Presently, several specific network process implementations are provided in MIIND: Wilson-Cowan and Ornstein Uhlenbeck type, and population density techniques for leaky-integrate-and-fire neurons driven by Poisson input. A design principle of MIIND is to support detailing: the refinement of an originally simple model into a form where more biological detail is included. Another design principle is extensibility: the reuse of an existing model in a larger, more extended one. One of the main uses of MIIND so far has been the instantiation of neural models of visual attention. Recently, we have added a library for implementing biologically inspired models of artificial vision, such as HMAX and recent successors. In the long run we hope to be able to apply suitably adapted neuronal mechanisms of attention to these artificial models.

1 The Philosophy of MIIND

1.1 The need for interoperability of models

Looking back on the developments in cognitive neuroscience over the last decade, the progress of experimental techniques is striking. Functional Magnetic Imaging Resonance (fMRI) machines have become commonplace and many of them are now purely dedicated to cognitive research. Techniques, such as EEG, PET, MEG and others have also matured and are increasingly used in conjunction with fMRI. Technological advances have allowed the development of novel ways of connecting electronic readout to neurons, leading to bigger multi-electrode arrays (MEAs) (Nicolelis et al., 2003; Navarro et al., 2005) or to neuron-on-silico chips (Fromherz, 2003; Schoen & Fromherz, 2007). This process seems to reinforce itself: the more we know about specific parts of the brain, the better we can plan the next experiment.

Enormous progress has also been made in the understanding of single neurons and neuronal systems. Researchers are beginning to explore large-scale models of biologically detailed networks (e.g., Djurfeldt et al., 2008). Relatively simple neuronal models now exist that model experimental data accurately, but which are much simpler than Hodgkin-Huxley type models (Brette & Gerstner, 2005). The BlueBrain project (e.g., Markram, 2006) aims to model an entire cortical column in realistic detail. The effects of synaptic plasticity rules

[1] Corresponding author, dekamps@comp.leeds.ac.uk
[2] Supported by CoTeSys grant no. 168 (`http://www.cotesys.org`)

can now be investigated in great detail both on the neuronal (e.g., Gerstner & Kistler, 2002) and the network level (Morrison, Aertsen, & Diesmann, 2007). But where experimental (cognitive) neuroscience has reached a breakthrough at the system level, and allows us to observe the global flow of neural activity in the active brain, computational modelling has yet to follow suit here. In general, progress in computational neuroscience has not informed models of high level cognition. In terms of understanding high level cognition, mental illnesses and endowing artefacts with human like intelligence we have not reached a breakthrough at the system level.

What is the cause for this? Although it is certainly true that means for modelling and theory building are not commensurate with the enormous investments in experimental equipment, in the opinion of the authors this is not the main reason. To include a realistic level of detail in models of high level cognition a lot of work will be necessary, which will involve a large number of people and expertise from various disciplines. However, in modelling we seem to have hit something of a complexity wall: most modelling is done on a project by project basis, in relatively small groups. Typically, the results of modelling are published, but publishing the implementation of the model, be it in the form of source code or an executable (or in other forms: e.g. CORBA/COM objects), is rare, although with the recent arrival of ModelDB (`http://senselab.med.yale.edu/modeldb`) this may slowly change. This means that the level of complexity of the model in general will not exceed that what can be achieved by a small research group (1-5 people) over a limited period of time (2-5 years). This is not a problem if one believes that the brain can be understood at a relatively high level of abstraction, and with relatively simple models, which only need to be scaled up to brain sized dimensions.

While it is not possible to disprove such an optimistic view, because of our lack of success in endowing artifacts with human- or animal-like intelligence, many researchers now adopt the view that we must invest more effort in understanding the brain and behaviour as a biological system (O'Reilly, 2006; Webb, 2001). This is a novel development in some disciplines, for example in Connectionism the need for biological realism used to be de-emphasized or even opposed. It has proven difficult to disentangle the neuronal level from the genetic level and the behavioural level from the neuronal level and modelling any higher level cognitive function involves a large number of brain areas, which are interconnected in complicated ways. The human brain will turn out to be extremely difficult to model. Some scientists argue that the way to go is to build brain- or biologically-inspired technological applications (Brooks, 1991; Pfeifer & Scheier, 1999; Webb, 2001), and learn from this how the brain functions. This is a valid approach, and will undoubtedly yield useful insights, but modelling the brain is also crucial to obtain insight into the causes of mental disorders and to improve existing human-brain interfaces systematically.

In this light, the complexity barrier imposed by small research groups and the limited amounts of time are a disaster. In order to make progress, it is necessary to construct models that can be reused by other research groups, which would reduce the enormous level of duplication which is currently going on. Ideally, models should be extensible in two directions: it should be possible to 'detail' them, replacing a certain coarse level of modelling by a finer grained one, and it should be possible to build complex models, using the current ones as building blocks. Not only would such a reuse of modelling greatly reduce duplication of programming efforts and allow the construction of complex models by small research groups, it would also greatly help in the transfer of expertise between the different disciplines involved in brain science.

Technically, it is extremely challenging to create such models, and the question is, what happens when they are there. Who maintains them? Who validates them? Who evaluates their usefulness? It is clear that coordination of some kind is necessary and it seems that with the creation of the International Neuroinformatics Coordinating Facility (INCF; `http://www.incf.org`), this is potentially forthcoming. However, even a coordination agency needs something to start with and it is clear that the first step must come from researchers in the field, since the problems sketched here surface most clearly when one tries to extend models and tries to incorporate work from other researchers. MIIND is an example of such efforts: starting as a relatively simple Artificial Neural Network (ANN) model for object-based attention (van der Velde & de Kamps, 2001), which was subsequently expanded into the Closed Loop Attention Model (CLAM) (van der Velde, de Kamps, & van der Voort van der Kleij, 2004), we extended it to include a neural blackboard architecture (van der Velde & de Kamps, 2006). We experienced first, that some of the programming work we had to do was quite repetitive and second, that we could not easily include our earlier work into the later, more sophisticated models. We also found that in these later models, which were characterised by a much more complex spatial organization, it was sometimes necessary to include more realistic descriptions of neuronal dynamics. Furthermore, we found it necessary to be able to integrate our models with those of others, in other words to construct models that are interoperable (see (Cannon et al., 2007) for a recent review of interoperability of neural simulators). For example, we are interested to see if the mechanisms described in (van der Velde & de Kamps, 2001) can be applied to recent models of object recognition in the ventral stream, such as HMAX and more recent variations (Riesenhuber & Poggio, 1999; Serre, Wolf, Bileschi, Riesenhuber, & Poggio, 2007). Finally, we have observed that models of others are often very similar in mathematical structure (Lanyon & Denham, 2004; Usher & Niebur, 1996; Hamker, 2005), even when expressing ideas or models quite different from our own.

We call our ideas on interoperability and extensibility of models "The Philosophy of MIIND" (de Kamps & Baier, 2007). They are realized in C++

code, but can be considered demonstrators, or realizations of design patterns (Gamma, Helm, Johnson, & Vlissides, 1994). Some of our ideas are not necessarily related to a C++ implementation and could, from a user's perspective also be realized on a GRID architecture.

1.2 The Philosophy of MIIND

The 'Philosophy of MIIND' is to isolate repetitive code tasks that occur during modelling as early as possible and to detach the problems that occur as much as possible from their modelling context. An example is SparseImplementationLib, which is a library for the representation of sparse random networks. Although initially created as an implementation for ANNs, it contains no explicit references to ANNs and is in fact quite useful in any problem that deals with sparse random networks. The idea is that a library has a limited, well defined task and that if the functionality of the library increases, it should be split.

Consider the example of a sparse network representation: most simulators have dealt with this problem, but their solution is usually not exposed to users and other developers. This is an opportunity for code reuse which is often missed. MIIND explicitly aims to expose its low level functionality. `SparseImplementation` (section 3) can be used for any problem which involves sparse irregular networks. DynamicLib (section 4.2) is a very generic solver for systems of equations, not necessarily restricted to neuronal simulations. Even if other developers decide not to adopt MIIND's implementation, they may get some ideas from the code. This is a major advance over algorithms which are published, but whose implementation is not publicly available.

MIIND's core functionality is implemented in C++, an object-oriented language which is suitable for high performance computing. Its object orientation is a major advantage over environments such as MATLAB, which rely on a functional programming paradigm. In our experience data structures in high level cognitive modelling become complicated and lead to a cumbersome implementation in MATLAB (van der Velde & de Kamps, 2001, 2006). The C++ template mechanism allows a static version of duck-typing (Koenig & Moo, 2005) which does not incur run time overheads. This means that central concepts can be expressed in a very abstract and powerful way that still leads to efficient code. In the population density methods that MIIND provides, this is of crucial importance. Also, a framework for equation solving must be efficient. This has motivated the choice for C++. At the same time the C++ syntax is obscure and complicated. Scripting languages such as Python offer a significant increase in productivity over C++ for code that is not time critical (estimated to be at least a factor 2, depending on the application (Prechelt, 2000)). At the moment we are equipping MIIND with a Python interface. LayerMappingLib already has such an interface, and a Python module will be built automatically when MIIND is compiled. The other libraries will receive a Python interface soon. MIIND uses ROOT (`http://root.cern.ch`) for visualisation. ROOT is Open Source software and is developed by CERN to process and visualize the data that will come from the LHC project. It offers scripting in C++ (CINT) and Python (PyRoot).

MIIND is OpenSource, released on a modified BSD license and hosted on SourceForge (`http://miind.sf.net`). The only modification with respect to the original license is that if you use MIIND in the preparation of a scientific publication, you must cite the 'currently valid reference', which is listed on MIIND's website.

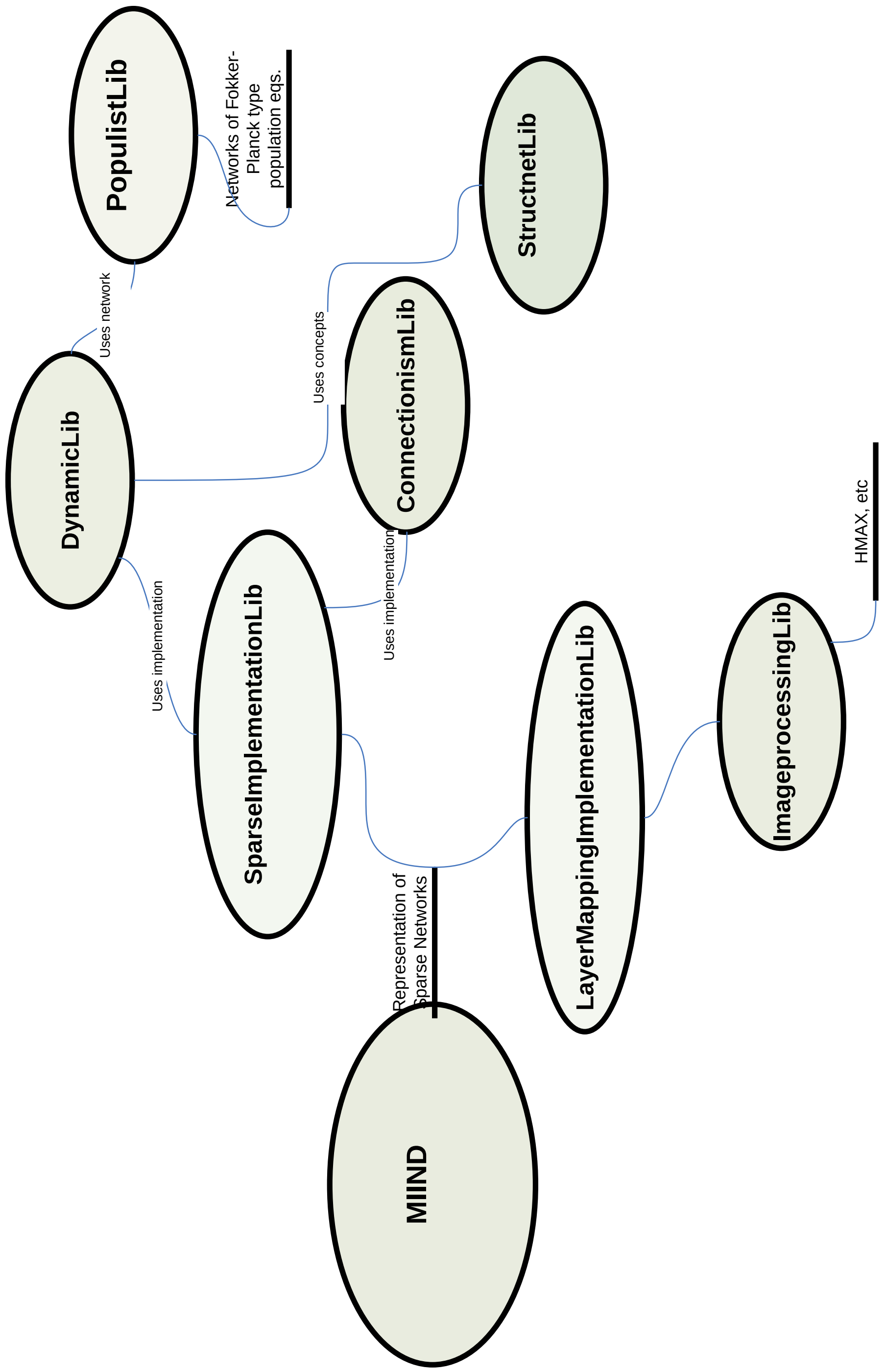
2 An overview of MIIND

Most concepts in MIIND are relatively high level and therefore not dependent on a particular programming language. Throughout this paper we will illustrate the concepts with simple code fragments of the 'Hello World' type. This code should be self explanatory, even to people who do not have a background in C++. Only in section 3 are some aspects of the C++ implementation discussed in more detail, but in a way that still should make sense to an audience with a general background in computer science. Throughout the text we will denote libraries, `class names` and *class methods* as shown here.

2.1 Sparse networks

An overview of MIIND can be found in Fig. 1. SparseImplementationLib is a library which offers generic support for the implementation of sparse random networks. Most biological networks are sparse: the number of nodes is typically large and the resources for the implementation of connections are typically limited. In the brain, for example, the 10^{11} nodes would lead to 10^{22} connections in a fully connected network. In practice, this number is closer to 10^{15}, so that an adjacency matrix representing the connection structure of a brain would be very sparse indeed. In neural simulations and other biological applications, the networks are also characterised by irregularity: the formation of connections may be driven by random processes or the networks may display little symmetry. Simulations of large networks are often limited by the amount of memory that is available and the adoption of an implementation which makes efficient use of memory is crucial to large network simulations.

Fig. 1. An overview of the MIIND libraries.



SparseImplementationLib supports the creation of large sparse irregular networks with non-trivial connection structures and provides facilities for reading them from and writing them to disk. Due to the use of C++'s template mechanism, SparseImplementationLib is extremely versatile (we elaborate on the relative merits of this mechanism in section 3) and some of the libraries that build on top of it (DynamicLib and PopulistLib) are demonstrators of the way in which it can be extended.

Although biological networks are often irregular, artificial models inspired by biology may display a high degree of symmetry. This is true, for example, for the Neocognitron (Fukushima, 1980) and also for more recent models of the ventral stream for object recognition (e.g. Riesenhuber & Poggio, 1999; Serre et al., 2007). Often, these networks are hierarchical structures of filters which display translation invariance. A literal network implementation of such a filter hierarchy, where the filter operations are implemented by connections, would require a large number of connections. Since the networks are typically feedforward, there are alternative implementations that are more efficient. The effect of a filter based network may be realized by storing a single filter and by applying that filter repeatedly to all locations in the previous layer. This reduces the large number of connections required to implement the whole parallel feature bank structure to a small set of filters that can be applied in rapid succession (or even in parallel given suitable parallelisation). This is the idea behind LayerMappingLib.

2.2 Simulating large systems of coupled equations

DynamicLib, the generic simulator, builds on top of **SparseImplementationLib**. It models large systems of coupled equations as networks, which is best illustrated by an example: suppose this network is modelled by Wilson-Cowan (Wilson & Cowan, 1972) equations:

$$\begin{aligned} \tau \frac{dE}{dt} &= -E + f(\alpha E - \beta I + \eta) \\ \tau \frac{dI}{dt} &= -I + f(\gamma E - \delta I + \eta), \end{aligned} \tag{1}$$

where E and I are the population firing rates of populations E and I respectively, η is an external input current contribution, α, β, γ and δ connectivity parameters and $f(x)$ is a so-called sigmoid function, the exact form of which can vary according to the model under consideration. τ is the population time constant, not to be confused with the membrane time constant of individual neurons. In the original work of Wilson and Cowan (1972), τ emerged as a free parameter, because they used a technique called time coarse graining.

Together with ANNs, Wilson-Cowan equations are one of the most widely used methods for cognitive neuroscience modelling (e.g., van der Velde & de Kamps, 2001; Lanyon & Denham, 2004; Usher & Niebur, 1996). Solving systems like Eq. 2 is not particularly difficult, but can be time consuming, especially if the system has to be set up from scratch.

DynamicLib offers facilities to set up the system of equations as a `DynamicNetwork`. For each population, a node is added, and for each connectivity parameter a corresponding edge. Each node is able to calculate the instantaneous weighted contribution from all other nodes. Node E, for example, is able to establish that its instantaneous external input is $\alpha E(t) - \beta I(t) + \eta$.

Each node is equipped with an `Algorithm`, which is responsible for maintaining and evolving the node's state. Several of these algorithms are already provided with MIIND. The algorithm `WilsonCowanAlgorithm`, for example, is able to solve equations of the type of Eq. 2 numerically. The user never needs to call numerical software directly, but simply configures the node with the appropriate algorithm.

Simulating a network is extremely simple. It entails the following steps:

- Create a `DynamicNetwork`.
- Create a `SimulationRunParameter`. Here parameters, such as the duration of the simulation, and the name of the file to which the simulation results should be written are specified.
- Create `DynamicNode`s, configure them with a `WilsonCowanAlgorithm` and add them to the network.
- Connect the `DynamicNodes` in the network, by specifying the connectivity parameters (efficacies).
- Configure the `DynamicNetwork` with the `SimulationParameter`.
- *Evolve* the `DynamicNetwork`. The network will drive the simulation automatically, and write the simulation results into a file. They can be analyzed at a later time, using the ROOT visualization package `http://root.cern.ch`, or any alternative.

The programming burden for creating and solving systems of Wilson-Cowan equations is greatly reduced in this way: there is no need to write or call numerical integrators, so the modeller can focus on setting up the network structure. The simulation is run by the network and the results can be written in a format that allows high quality visualization (see section 4.2). In (van der Velde & de Kamps, 2006) we discussed a large network that we were able to develop in a matter of days (Fig. 6).

As a demonstration of the versatility of DynamicLib, we follow some of the modelling work done by Amit and Brunel (1997b, 1997a). They used a clever method to describe the steady state activity of large populations of leaky-integrate-and-fire (LIF) neurons and applied them to a neuronal model of working memory. We will demonstrate how a relatively simple extension of the `WilsonCowanAlgorithm`, the `OrnsteinUhlenbeckAlgorithm` (or `OUAlgorithm` for short) can quickly replicate and extend the networks studied by Amit and Brunel (1997b).

DynamicLib is not restricted to sets of Wilson-Cowan equations. Several other algorithms have been provided, including so-called population density algorithms (de Kamps, 2003, 2006). These techniques have been developed to model the response of large populations of spiking neurons (Knight, Manin, & Sirovich, 1996; Omurtag, Knight, & Sirovich, 2000; Nykamp & Tranchina, 2000; Haskell, Nykamp, & Tranchina, 2001; Apfaltrer, Ly, & Tranchina, 2006; Muller, Buesing, Schemmel, & Meier, 2007). They are akin to coupled systems of Fokker-Planck equations and describe approximately the same dynamics although they are more generally applicable. They describe transient neuronal dynamics much more accurately than Wilson-Cowan equations, at the expense of being computationally intensive and more difficult to implement. Since the implementation is provided with MIIND, setting up such a system of equations is about as simple as setting up a network as described above and involves largely the same steps. PopulistLib provides these algorithms (see section 5).

There is an important link between `OUAlgorithm` and population density techniques in that the former describes steady states of the latter. `OUAlgorithm` takes much less computing time than population density techniques. So, one may explore a network and find suitable network parameters by using `OUAlgorithm`. Once the network is fixed in a structure that gives the desired steady state activity, a single programming statement exchanges `OUAlgorithm` for a population density algorithm. The same simulation can now be run, but with a much more realistic description of transient neuronal dynamics (and at the expense of much more computing time). *Modelling a network and modelling the neuronal dynamics in the network can be done completely independently from each other.* It is even possible to create heterogeneous networks, where a large part of the network is described by a simple efficient algorithm and a subset of the simulations are performed in much greater detail.

`Algorithms` in principle could be constructed from other simulators, for example, compartmental models or point model neurons, as simulated by NEURON, GENESIS and NEST (see (Brette et al., 2007) for a recent overview of publicly available spiking neuron simulators). DynamicLib is a framework, because it allows the insertion of new `Algorithm`s and visualization methods. We believe that this framework demonstrates one possible design pattern for models that are interoperable. Since this is an important topic, we will come back to it in the discussion.

`StructNetLib` offers support for endowing networks with a spatial structure.

In some neuroscience models the spatial structure of the network is just as important as the connection structure and in some cases the network is even defined in terms of its spatial structure, for example when connections are restricted to certain distances or when neurons have a receptive field which is spatially restricted.

`ConnectionismLib` contains a few basic algorithms, such as Hebbian training and back-propagation of errors, which can be useful in the construction of network models, but this part of MIIND is not well developed. Those who are looking for libraries which offer a large variation of connectionist algorithms are explicitly referred elsewhere (e.g., O'Reilly & Rudy, 2001). The reason why we have included this library in the public release is that it provides a nice illustration of how SparseImplementationLib can be used to support sparse neural networks. Also, some of our earlier models (van der Velde & de Kamps, 2001) were created with this code and it is possible to replicate them.

3 SparseImplementationLib

3.1 The sparse network memory model

Any network is a collection of nodes connected by edges. Typically, there are numerical values associated with the edges, and often a computer implementation consists of a vector for storing the values of the nodes and a so-called adjacency matrix (or weight matrix) for storing the values of the edges (or weights). This is a very general representation for networks, which allows fast access to both the edge values and the node values, and is easy to program. This representation is quite wasteful, however, when the network is sparse, i.e. when most of the entries in the weight matrix are zero. Biological networks are typically large and sparse (since the connections must be physically implemented) and realistic simulations, relying on such a network representation would quickly run out of memory.

Biological networks are usually irregular as well. Symmetries, such as translation or rotation invariance which could lead to clever representations of the networks are usually realized only approximately in biological networks and often there is no other option than to explicitly represent each edge. But non-existing connections need not be represented! The key idea for an efficient representation of an irregular sparse network is illustrated in Fig. 2.

Each node has the responsibility for representing its numerical value (or activation). On top of that, the node maintains a list of connections. Each connection is a pointer-value pair. The pointer is a reference to a node which is connected

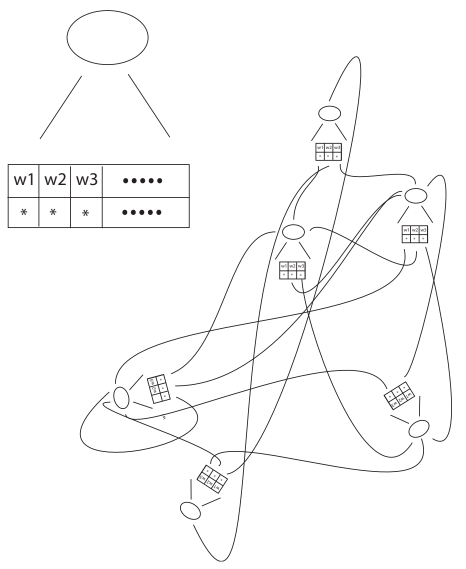


Fig. 2. A C++ data structure for representing sparse networks. Each node contains a list. The list contains pointer-weight pairs. The pointer points to a node with which this one is connected and the weight represents the strength of the connection. A collection of such nodes represents a network. Reprinted with permission from (de Kamps & Baier, 2007).

to this node, its predecessor[3]; the value is the numerical value of the edge (weight) connecting the predecessor node to this one. So a node keeps a list of all predecessor nodes (and weights) and is thereby responsible for maintaining a list of which nodes are connected to it. It is clear that a collection of such nodes form an implicit network representation. It is an implicit network representation because there is no information about the network associated with the nodes; information on a network must therefore be obtained by visiting each node, which may be impractical.

An explicit network representation entails the grouping of nodes that make up a network in a common class. This has two advantages: first, this makes it possible to query the network at its properties (how many neurons? how many input neurons? etc.). Second, it disambiguates the concept of a network (a single node on its own might belong to two networks, in principle). Also, it allows the definition of the notion of consistency of a network (if a node contains a reference to a predecessor node that is not in the network, the network is not consistent).

So, networks may be represented by lists of nodes, where each node itself maintains a list of its predecessors and numerical values (weights) associated with each predecessor. Below, we will discuss a C++ implementation of these

[3] This terminology originates from the ANN past of SparseImplementationLib, where input neurons are really predecessors, the terminology is arbitrary, however and we could have equally called them successor nodes

ideas.

3.2 A C++ implementation - Basic concepts

In this section we discuss some details of the C++ implementation. An abstract class, `AbstractSparseNode`, implements the central concept of a node that is aware of its neighbouring nodes in the network. An `AbstractSparseNode` stores a numerical value, whose type is determined by a template argument (usually a `double`). This value can be interpreted as the nodes activation value. It can be set and read by *GetValue()* and *SetValue()* respectively. Furthermore, an `AbstractNode` maintains a list of so-called `Connections`. A `Connection` consists of a pair of a pointer to another `AbstractNode` and a weight value. So a single `Connection` represents an edge in the network. The *PushBackConnection()* allows new `Connections` to be added to the list of `Connection`s that each `AbstractSparseNode` maintains internally. The *InnerProduct()* method computes the scalar product of the input contributions to the node. Normally, this amounts to the standard weighted sum of the activities at its input: if a node j has other nodes i, $i = 0, .., n-1$ as input (where n is the number of inputs), it can calculate:

$$a_i = \sum w_{ij} a_j, \tag{2}$$

where a_i is the activity of node i and w_{ij} is the weight from edge $j \rightarrow i$.

Because each `AbstractSparseNode` has a pointer to each predecessor, it can retrieve its predecessor's activity and weight it appropriately. The weight is often a floating point value, but since the type of the weight is a template argument, more complicated types can be used. In section 4.3, we will give an example of a network, where links are determined by two numbers, instead of one. Clearly, the notion of a weighted sum of inputs, the result of *InnerProduct()* must then be redefined. The fact that both the types of the activation value of the nodes and the weights of the network can be parameterised by specific choices for the template argument, contributes considerably to the extensibility of SparseImplementationLib.

3.2.1 `SparseNode`

A `SparseNode` derives from `AbstractSparseNode`. `SparseNode` is a concrete data type and can also maintain a reference to a squashing function. Such nodes are able to calculate

$$a_i = f(\sum w_{ij} a_j),$$

where f is the squashing function, usually a sigmoid (predefined sigmoids come with MIIND, so the user merely has to configure the node). Many users will

```
// Example code for connecting nodes to each other.
// Illustration purposes only ! Don't code like this !
// Use SparseImplementation.

        D_SparseNode node_1, node_2, node_3;

        // set Ids
        node_1.SetId(NodeId(1));
        node_2.SetId(NodeId(2));
        node_3.SetId(NodeId(3));

        // create two connections: 2->1 and 3->1
        pair<D_SparseNode*,double> connection_12(&node_2, 2.0);

        pair<D_SparseNode*,double> connection_13(&node_3,-2.0);

        // add the Connections
        node_1.PushBackConnection(connection_12);
        node_1.PushBackConnection(connection_13);

        // Set activation in Node 2 and 3
        node_2.SetValue(1.0);
        node_3.SetValue(1.0);

        if ( node_1.InnerProduct() != 0 )
          // this would be an error
                return false;

        // We can give a node a squashing function:
        Sigmoid sigmoid;
        node_1.ExchangeSquashingFunction(&sigmoid);
        return true;
```

Fig. 3. Code that demonstrates the creation and linking of nodes. Although this represents a network implicitly, users are recommended to use `SparseImplementation` instead.

recognize that everything now is in place to represent ANNs. Below we will give some example code for creating a small neural network. Indeed, SparseImplementationLib was originally developed with ANNs in mind. The squashing function is the only explicit reference to the classnameSparseNode's ANN past. We will show in applications of `DynamicLib`, that very generic network processes can be modelled. In Fig. 3 we give example code for the creation of a network.

The concept of an `AbstractSparseNode` generalizes well. We will present an example in section 4.2. C++'s template mechanism has been rightly criticised as obscure and cumbersome. It provides a static form of duck-typing (Koenig & Moo, 2005) and indeed dynamic versions of duck-typing, such as implemented in Python, are much more transparent to the user. It is difficult to see, however, how the concept of an `AbstractSparseNode` can be implemented in e.g. Python whilst retaining its efficiency. Consider, for example, the computation of the inner product that `AbstractSparseNode` provides. The strong typing maintained by the template mechanism ensures that at compile time the `operator+()` and `operator*()` versions which are necessary to compute the inner product are determined. This allows them to be inlined, eliminating the overhead of a function call, which will be made for *every* weight that is used to compute the inner product. A similar construct in Python will be reasonably efficient for built-in types, but will have to perform a function call for every user-defined type weight. Due to the dynamic resolution of function arguments in Python, it is hard to see how this can be done efficiently. Since the calculation of inner products can be a bottleneck in scientific computation, we decided that `AbstractSparseNode` is a class for which this efficiency argument can not be ignored.

3.2.2 `SparseImplementation` *and* `Architecture`

As the code shows, using the nodes is straightforward. We show it, because it gives insight in how the code can be used and extended. These nodes constitute a very simple neural network. There are several reasons not to code like this, however, (unless one wants to develop a novel implementation). In the first place, this network does not have an explicit representation: as argued above, it is a collection of nodes that implicitly represent a network. The class `SparseImplementation` is an explicit representation of a network: one can ask it, how many nodes there are, or to write itself to disk. The second reason to use `SparseImplementation` is that it isolates the user from the pointer representations at the node level. Copying a `SparseImplementation` is safe: all internal pointer values are correctly updated without the user having to keep track of the details. Finally, building a large network in the way shown in Fig. 3 can be laborious. In particular if the networks have a complicated structure, one may want to automate aspects of its creation. The way this is done, is to create an `Architecture`, which is a mathematical description of a network. `SparseImplementation` accepts an `Architecture` as a constructor argument.

An `Architecture` is intended as a mathematical shorthand for a network's architecture. For example if a network is fully connected, it suffices to specify the number of nodes. If a network is a fully connected feedforward network, it suffices to give the number of nodes in each layer. In the former case an `Architecture` must be created, in the latter case a `LayeredArchitecture`. Sometimes `Architecture`s are not sufficient to characterise the network and a list of connections must be provided. Some example code can be found at `http://miind.sf.net/examples_nn_2008`.

3.3 A C++ implementation - Advanced concepts

3.3.1 Navigation through the network

In many cases the need for users to directly access node values or weight values can be reduced. In connectionist networks, users can read in and read out patterns, whereas a training algorithm determines the weight settings. In DynamicLib, users create the network using suitable class methods and the visualization of activation values in the network is done by so-called `Handler`s. Wherever possible we try to avoid the need for direct access to weights and activation values in the network. Sometimes, however, this can not be avoided: somebody who writes a training algorithm for connectionist networks, for example, needs direct and efficient access to the weights and activation values.

A `SparseImplementation` provides *iterators* that give direct access to nodes. The nodes themselves provide iterators to their connection list. Because iterators are a relatively advanced C++ concept, we give an example of access on the website: `http://miind.sf.net/examples_nn_2008`, for those who want to get a feeling for how to use them in a `SparseImplementation`.

3.3.2 Reversing the network

Usually, the concept that each node maintains a list of its predecessors is sufficient for an efficient network representation. However, `SparseImplementation` also supports connections to successors rather than predecessors. If a node simulates a spiking neuron, for example, the spike must be delivered to successor neurons rather than predecessors. This difference is semantic, however: the key idea of SparseImplementationLib is that each node maintains a list of nodes to which it is connected and whether this connection denotes a predecessor or a successor relationship is immaterial.

There are situations, however, where a network in normal circumstances needs to know about its predecessors only, but in special circumstances also about its successors. Backpropagation is such a situation. The backpropagation algorithm requires information transfer in the network in two directions: during normal operation information flows from input to output, but during training information must flow back from output to input!

How does one deal with this? At the level of a `SparseImplementation` it is possible to establish which node is the successor of which other node, given the fact that all predecessor relations are defined already. It would then be possible to insert into each node a list of successors, complimentary to the already existing list of predecessors. `ReversibleSparseNode` is derived from `SparseNode` and has facilities to maintain this extra list. Hence, if a `SparseImplementation` is instantiated with `ReversibleSparseNodes`, rather than normal `SparseNodes`, `SparseImplementation` is able to insert the successor relations into a node, on top of the already existing predecessor relations. Information can flow in two directions in such a network.

One aspect of `SparseImplementation` is that the architecture must be known in advance. We found this somewhat inflexible in some cases, and in DynamicLib, we introduce `DynamicImplementation` which allows adding nodes and connections on the fly. We will provide an example of this in section 4.2.

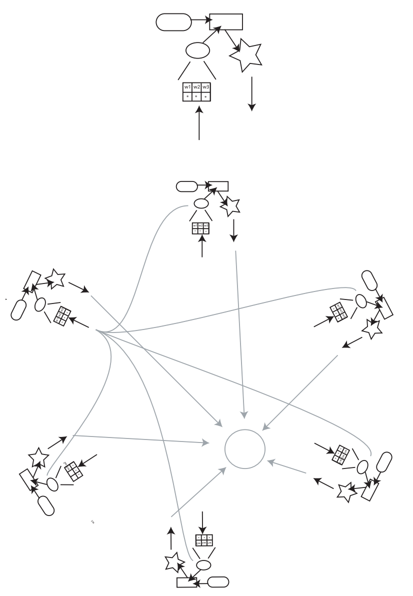


Fig. 4. A single DynamicNode is derived from a SparseNode. It has an AbstractAlgorithm (oval), which operates on a NodeState (rectangle). When prompted by the simulation loop, the AbstractReportHandler sends the current NodeState to a central file. DynamicNodes are almost autonomous. The central simulation loop determines which Node is in line to evolve its NodeState over a short time step, but the Nodes themselves collect input from other Nodes and deliver this to their own Algorithms which evolve the Node's NodeState. This setup is easy to parallelise. Reprinted with permission from (de Kamps & Baier, 2007).

4 DynamicLib

4.1 Introduction

`DynamicNode`s derive from `AbstractSparseNode`s. They represent a significant extension: not only can they query other nodes for their activities, they also contain a reference to an `AbstractAlgorithm`, which in turn maintains a `NodeState`. The key idea is that `DynamicNode`s can evolve their `NodeState` by requesting their `Algorithms` to do so: `DynamicNode` has an *Evolve()* method, which calls its `Algorithm`s *Evolve()* method. The `NodeState` takes over the role that the activation value had in `SparseNode`. The `NodeState` describes the state of the node at a certain time t and the `Algorithm`'s *Evolve()* method evolves that node's state over a time Δt, which is usually small. `DynamicNodes`, like `AbstractSparseNodes` maintain a list of nodes that connect to them with a weight for every connection. At every time t, they are able to evaluate the instantaneous contribution of other nodes to itself and that input is passed to the node's `Algorithm` as a parameter.

At the highest level `DynamicNetwork`'s *Evolve()* method initiates a loop over all nodes, the simulation loop, in which it requests that every `DynamicNode` evolve itself over a short period of time. The `DynamicNetwork` does this repeatedly and in such a way a simulation of the network dynamics emerges. A `DynamicNode` is also configured with a `ReportHandler`. At fixed times, the simulation loop queries the `DynamicNodes` for a `Report`. The `ReportHandler` of the `DynamicNode` delivers the `Report` and the `Reports` are written to disk so that a record of the simulation is produced. Also, the simulation loop maintains a log file to indicate how far the simulation has progressed and to keep a record of exceptional conditions that occurred during simulation. In Fig. 4 we show a graphical representation of the classes involved in `DynamicNetwork`. In the next section we will present a Wilson-Cowan equation model as a concrete example of the abstract concepts described in this section.

4.2 Modelling Wilson-Cowan equations

Consider a network which consists of two populations, one of which is described by Eq. 2

$$\tau \frac{dE}{dt} = -E + f(\alpha E + \epsilon \nu), \tag{3}$$

and one of which simply maintains a fixed output rate and serves as an external population to the network.

In Fig. 5 we show how such a network is configured. First, the `WilsonCowanAlgorithms` is defined and configured with the appropriate `WilsonCowanParameter` parameter, which defines the sigmoid parameters. A network also needs input: therefore a `RateAlgorithm` is created, an algorithm whose only action is to set the `NodeState` of the `DynamicNode` to which it belongs to a fixed rate (the `NodeState` consist of a single floating point value in this case). The nodes are then created in the `DynamicNetwork`, with their own copy of the `WilsonCowanAlgorithm` (or `RateAlgorithm`). A user receives a `NodeId` as a reference to `DynamicNode` that was just created in the `DynamicNetwork`. These `NodeIds` can then be used to define `Connections` in the network. After the definition of the `DynamicNetwork`, one only has to *Configure* it and to *Evolve* it.

In this code a standard sigmoid is used: a function of the form:

$$f(x) = \frac{f_{max}}{1 + e^{-\beta x}},$$

```
// define a D_Network, a network whose weights are doubles
D_DynamicNetwork network;

Time tau = PARAMETER_NEURON._tau;
Rate rate_max = 100.0;
double noise = 1.0;

// define some efficacy
Efficacy epsilon = 0.1;

// define some input rate
Rate nu = 10;

// Define a node with a fixed output rate
D_RateAlgorithm rate_alg(nu);
NodeId id_rate = network.AddNode(rate_alg,EXCITATORY);

// Define the receiving node
WilsonCowanParameter par_sigmoid(tau,rate_max,noise);

WilsonCowanAlgorithm algorithm_exc(par_sigmoid);
NodeId id = network.AddNode(algorithm_exc,EXCITATORY);

// connect the two nodes
network.MakeFirstInputOfSecond(id_rate,id,epsilon);

// define a handler to store the simulation results
RootReportHandler
        handler
        (
                "test/wilsonresponse.root",     // simulation results
                false,                     // do not display on screen
                true                       // write into file
        );

SimulationRunParameter
        par_run
        (
                handler,                   // the handler object
                1000000,                   // maximum number of iterations
                0,                         // start time of simulation
                0.5,                       // end time of simulation
                1e-4,                      // report time
                1e-4,                      // update time
                1e-5,                      // network step time
                "test/wilsonresponse.log"    // log file name
        );

bool b_configure = network.ConfigureSimulation(par_run);

bool b_evolve = network.Evolve();
```

Fig. 5. Example of setting up a Wilson-Cowan network, consisting of one population and an external current.

where f_{max} is the maximum response of the node and β is a noise parameter. In the code of Fig. 5 it can be seen how these parameters are set. The procedure follows closely the steps outlined in section 2.2. There is no need to call or define numerical integrators from the user's point of view.

Setting up large networks is a trivial exercise. It just comes down to using *AddNode* and *MakeFirstInputOfSecond* repeatedly. A very large network which was used to model a neuronal architecture for compositional representations (van der Velde & de Kamps, 2006) is shown in Fig. 6.

4.3 Modelling steady states of spiking neurons: dyadic connections

Wilson-Cowan equations are perhaps the most widely used modelling technique in large-scale network modelling. It has been shown that Wilson-Cowan

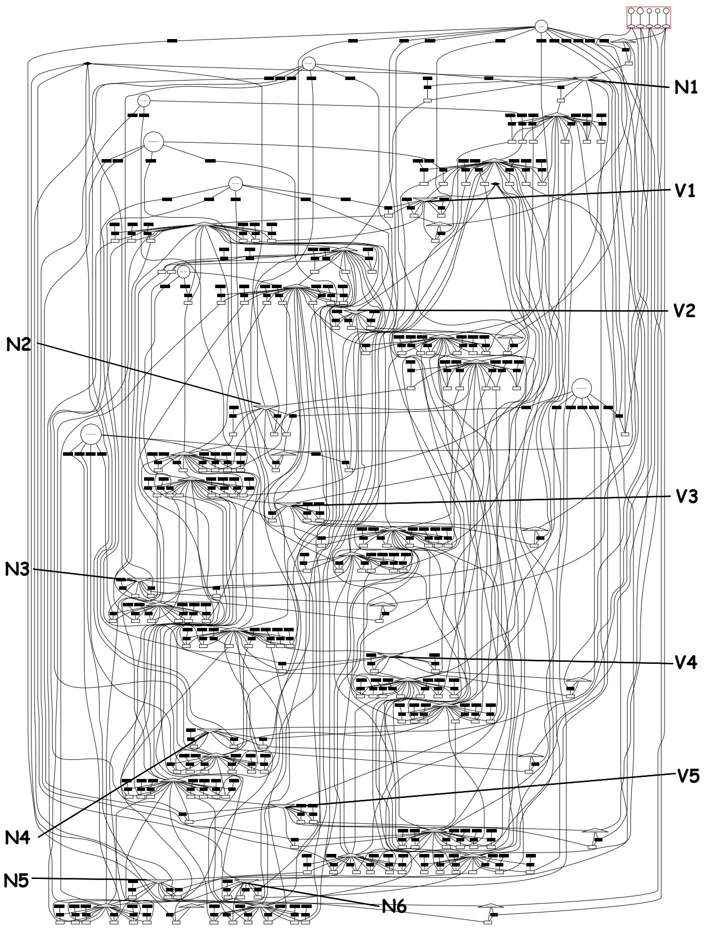


Fig. 6. A large network simulated with DynamicLib.

equations adequately describe the trend in the activity of a group of neurons reasonably well (Gerstner, 1995), but not the transient dynamics[4]. Also, Wilson-Cowan techniques contain a sigmoid for which the original motivation (Wilson & Cowan, 1972) is not considered to be valid anymore. Amit and Brunel (1997b) introduced a modelling technique which describes networks of spiking (LIF) neurons in terms of their steady state. It is possible to derive these equations from first principles, under some reasonably plausible assumptions about connectivity and firing rates in cortex. So states in these networks actually describe steady-state activity of networks of spiking LIF neurons as accurately as direct simulations would.

[4] This is due to a procedure called time coarse graining, which was used to reduce integral equations to the kind of differential equations that are nowadays commonly referred to as Wilson-Cowan equations

As we will show, the response of a neuronal population is not only determined by the average input, as in ANNs and in Wilson-Cowan equations of the kind discussed above, but also by the *variability* of the input. This reflects the fact that the input is assumed to consist of stochastic spike trains that are assumed to be Poisson distributed. Clearly, this is a step up in neuronal realism. This is one reason for demonstrating the solution of these equations with DynamicLib. Another important reason is that it demonstrates the flexibility of DynamicLib in handling connections of any kind: as we will see connections for these kind of networks are dyadic, a single connection is determined by two floating point numbers. The first number is the average efficacy from a neuron in one population to another and the second is the effective number of connections between the two populations. DynamicLib is able to handle connections of any type due to C++'s template mechanism. Although this is an important issue, this is quite technical and we refer the C++ aficionados to `http://miind.sf.net/examples_nn_2008`.

We will now present the equations used by Amit and Brunel (1997b):

$$\nu_i = \phi_i(\mu_i, \sigma_i), \tag{4}$$

where:

$$\phi_i(\mu_i, \sigma_i) \equiv \left\{ \tau_{ref,i} + \sqrt{\pi}\tau_i \int_{\frac{V_{reset,i}-\mu_i}{\sigma_i}}^{\frac{\theta_i - \mu_i}{\sigma_i}} du \, [1 + \mathrm{erf}(u)] \, e^{u^2} \right\}^{-1} \tag{5}$$

$$\begin{aligned} \mu_i = & \qquad \tau_i \sum_j J_{ij} N_{ij} \nu_j, \\ \sigma_i = & \qquad \sqrt{\sum_j \tau_i J_{ij}^2 N_{ij}}. \end{aligned} \tag{6}$$

τ_i, $\tau_{ref,i}$ are the membrane time constant and the absolute refractory period, respectively, in s, θ_i and $V_{reset,i}$ the threshold potential and the reset potential, respectively, in V, all for neurons in population i. N_{ij} is the effective number of neurons from population j seen by a neuron in population i and J_{ij} the average efficacy from a spike in population j on a neuron in population i in V. These equations form a closed system which can be solved for ν_i. In practice, one does this by introducing a pseudo-dynamics:

$$\tau_i \frac{d\nu_i}{dt} = -\nu_i + \phi(\mu_i, \sigma_i), \tag{7}$$

and selecting initial values $\nu_i(0)$ (Renart, Brunel, & Wang, 2004; la Camera, Rauch, Lscher, Senn, & Fusi, 2004; de Kamps, 2005).

For `OUAlgorithms` it is necessary to introduce a novel inner product concept: the input rates from other nodes are given by the rates ν_j. These rates are

```
// Note we now need an OU_Network instead of a D_Network
OU_Network network;

Potential sigma = 2e-3;
Potential mu    = 20e-3;

Time tau = PARAMETER_NEURON._tau;
Rate nu = mu*mu/(sigma*sigma*tau);
Rate J = sigma*sigma/mu;

OU_Connection
        con
        (
                1,
                J
        );

// Define a node with a fixed output rate
OU_RateAlgorithm rate_alg(nu);
NodeId id_rate = network.AddNode(rate_alg,EXCITATORY);

// Define the receiving node
OU_Algorithm algorithm_exc(PARAMETER_NEURON);
NodeId id = network.AddNode(algorithm_exc,EXCITATORY);

// connect the two nodes
network.MakeFirstInputOfSecond(id_rate,id,con);

// define a handler to store the simulation results
RootReportHandler
        handler
        (
                "test/ouresponse.root", // simulation results
                false,           // do not display on screen
                true             // write into file
        );

SimulationRunParameter
        par_run
        (
                handler,         // the handler object
                1000000,         // maximum number of iterations
                0,               // start time of simulation
                0.1,             // end time of simulation
                1e-4,            // report time
                1e-4,            // update time
                1e-5,            // network step time
                "test/ouresponse.log"  // log file name
        );

bool b_configure = network.ConfigureSimulation(par_run);

bool b_evolve = network.Evolve();
```

Fig. 7. The code for the simulation of a single population network, described by the dynamics of 7.

converted by application of Eq. 6 to two-tuples $(N_{ij}, J_{ij})^T$ and result in another two-tuple: $(\mu, \sigma)^T$. So the inner product is dyadic.

This example shows that the concept of inner product can deviate from the standard inner product and that it acquires a specific meaning in terms of the `Algorithm` that is used by the `DynamicNode`. The `Algorithm` determines the type of connection that the node needs and this in turn determines what kind of inner product will be evaluated at the node. The fact that the type of the connection is a template argument enables MIIND to introduce novel inner product concepts at any stage.

It is instructive to simulate a network of one population first. The code is presented in Fig. 7.

There are only minor changes with respect to the Wilson-Cowan simulation of section 4.2:

- Instead of a `D_DynamicNetwork`, an `OU_DynamicNetwork` is used. The consequence of this is that the connections in the network are so-called `OU_Connections`, which are structs that can hold two floating point numbers. Also, the algorithm uses the dyadic inner product of Eq. 6 rather than the standard inner product of Eq. 2. All these changes are triggered by the choice for `OU_DynamicNetwork` and are, from the user's point of view, automatic.
- Instead of a `SigmoidParameter`, the `OU_Algorithm` must be configured with an `OUParameter`, which determines the properties of the neurons to be simulated, such as the membrane time constant, the threshold value, etc.
- An `OU_Connection` is created, which is defined by two numbers and this object is then used in the network's *MakeFirstInputOfSecond* method.

The single population receives external input from another one, the 'fixed' population which maintains a fixed firing rate. Assuming that there is effectively one ($N = 1$) connection between the two populations, one can choose J, the efficacy of the connection between the populations and ν, the firing rate of the 'fixed' population and one can calculate μ and σ using Eq. 6. From them, using Eq. 5, the firing rate of the population receiving the input can be calculated. Alternatively, one can specify μ and σ since they determine ν and J according to:

$$J = \frac{\sigma^2}{\mu} \tag{8}$$

$$\nu = \frac{\mu^2}{\sigma^2} \tag{9}$$

This is what was done in the simulation shown in Fig. 8: μ was taken to be 15 mV and σ to be 2 mV and a J and ν that would lead to such μ and σ were then used as parameters for the simulation. Keeping σ fixed and varying μ gives a so-called $f-I$ or gain function curve (Ricciardi, 1977; Amit & Tsodyks, 1991). On the right hand side of Fig. 8 we show a $f - I$ curve, calculated by Eq. 5. On the left hand side we show simulations of the network for various μ values and it can be seen that the firing rate of the population indeed asymptotes towards the corresponding value of the $f - I$ curve.

In Fig. 9 we show a network of two populations: one excitatory and one inhibitory, both driven by an external input current. The steady state is given

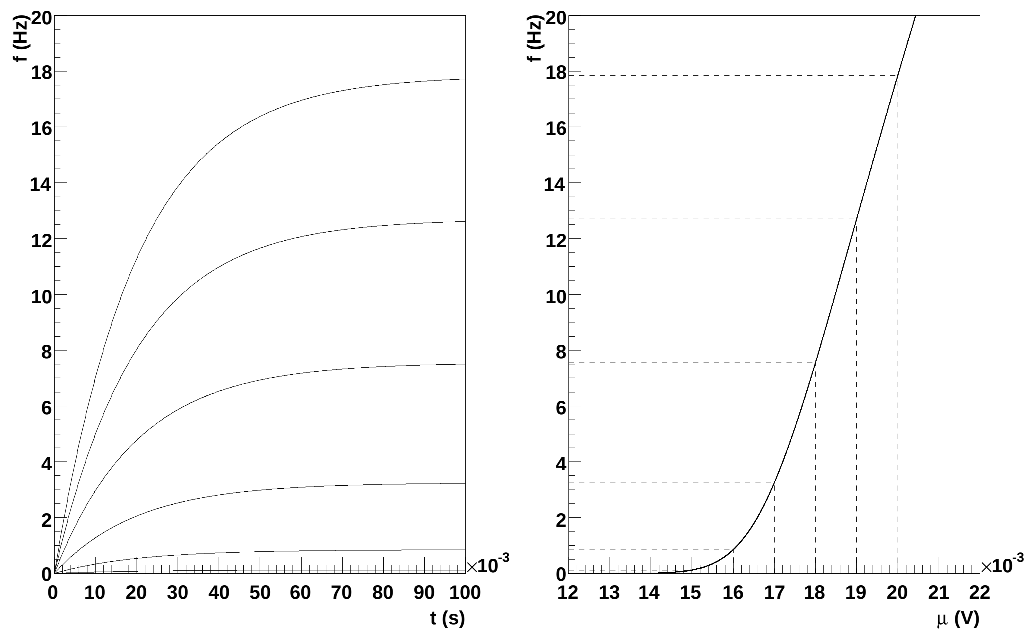


Fig. 8. The simulation results for a population which receives input with a fixed ν and J. In the simulations ν and σ where chosen such that σ was held fixed at 2 mV and μ at 15, 16, 17, 18, 19 and 20 mV, respectively. The firing rate of the population (left) quickly converges to rates given by the $f - I$ curve for the μ and σ values (right).

by Eq. 4. It is instructive to write these equations out (Amit & Brunel, 1997b):

$$\begin{aligned}
\mu_e &= \tau_e(N_E J_{EE}\nu_e - N_I J_{EI}\nu_i) \\
\sigma_e^2 &= \tau_e(N_E J_{EE}^2\nu_e + N_I J_{EI}^2\nu_i) \\
\mu_i &= \tau_i(C_E J_{IE}\nu_e - N_I J_{II}\nu_i) \\
\sigma_i^2 &= \tau_i(N_E J_{IE}^2\nu_e + N_I J_{II}^2\nu_i) \\
\nu_e &= \phi_e(\mu_e, \sigma_e) \\
\nu_i &= \phi_i(\mu_i, \sigma_i)
\end{aligned} \tag{10}$$

This shows that this deceptively simple simulation corresponds to the solution of a complex system of equations. This is even more obvious if the network contains more than two populations: the equations in (Brunel, 2000), which describe a network of four populations, are long and complex. If these equations were coded directly, which would be a tedious job, it would be very difficult to detect errors in the code. In the DynamicLib the simulation of such a network would merely entail adding more nodes and connections.

The $f - I$ curve is derived from first principles: it is given by calculating the average first exit time for an Ornstein-Uhlenbeck process with an absorbing boundary (the neuron threshold) (Gardiner, 1997; van Kampen, 1997). It has been shown to give a very good approximation of the firing rate of a large

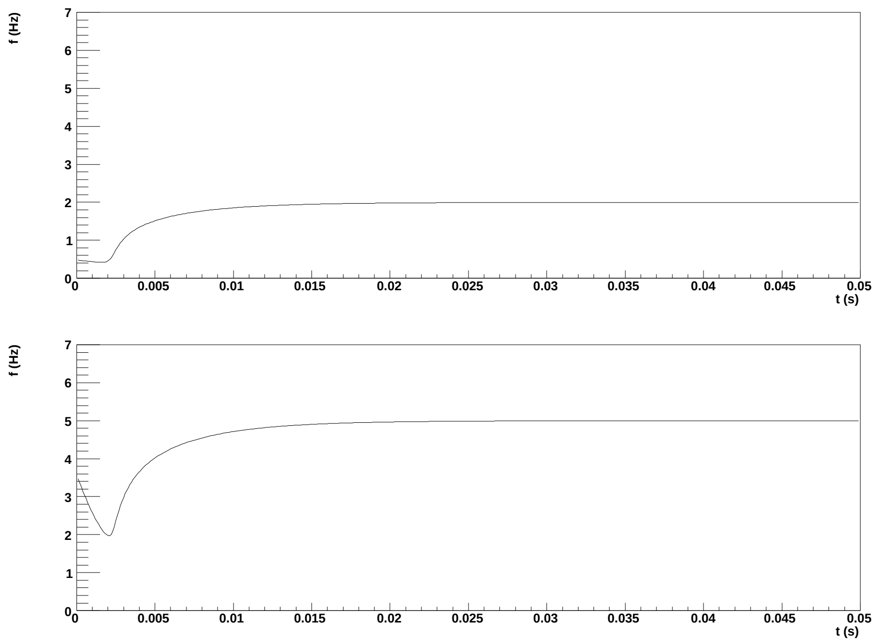


Fig. 9. A simulation of a two population network. An excitatory (above) and an inhibitory (below) population are connected to each other and to themselves and driven by an external input. The inhibitory population has a smaller population time constant, so initially network activity is suppressed. Once the excitatory population kicks in, the network activity quickly converges to a steady state. The qualitative behaviour of the network activity is similar to that of a comparable network described by Wilson and Cowan (1972).

population of spiking LIF neurons when they receive a constant input at high input rates and small synaptic efficacies. Under these conditions the stochastic spike train input may be replaced by a Gaussian white noise defined by μ and σ. Extensive simulations have shown that networks of large LIF populations indeed settle at the firing rates given by Eq. 5 (Amit & Brunel, 1997a; Brunel, 2000). Applications of Eq. 5 can be found in the description of delay dynamics, which is believed to be the neuronal substrate of working memory (e.g., Amit & Brunel, 1997b, 1997a; Brunel, 2000; de Kamps, 2005).

5 PopulistLib: towards networks of spiking neurons.

It is important that no misunderstanding arises about the title of this section: at present MIIND does not contain simulators for groups of spiking neurons. There are plenty of packages around that are very suitable for the direct simulation of compartmental model of neurons or networks of leaky-integrate-and-fire (LIF) neurons (Brette et al., 2007). MIIND does contain algorithms for population density techniques, however, and they can describe the behaviour of large groups of LIF neurons very accurately: often these techniques deliver the same information as straightforward simulations of LIF neurons, but much more efficiently (Omurtag et al., 2000; Nykamp & Tranchina, 2000). Another reason why these techniques are important is that they can be used together with standard techniques for analysing dynamical systems. Successful analyses of the phase space portrait of simple systems have been given in (Brunel & Hakim, 1999; Mattia & Del Giudice, 2002; Sirovich, Omurtag, & Lubliner, 2006). This is an exciting and important development: for direct simulation a large number of neuronal and network parameters must be chosen. In particular the network parameters, such as synaptic efficacies, are not well constrained experimentally and an overwhelming freedom of choice exists in direct simulation. Knowing the phase space portrait of a circuit reduces the enormous freedom in the choice of parameters to the choice of a much smaller number of dynamic regimes. It then becomes much easier to motivate the choice of simulation parameters. A final important reason to consider these techniques is the fact that they link the individual neuronal level with the network level. It has been amply demonstrated that groups of spiking LIF neurons are adequately described by population density techniques (Omurtag et al., 2000; Nykamp & Tranchina, 2000). So by connecting several populations into networks, it becomes possible to model large networks of such spiking neurons. We will give a demonstration in this section of a simple network of this kind. We will emphasize that from the perspective of the user, the code is almost identical to that of examples in previous sections.

Population density techniques are partial differential equations, usually Fokker-Planck equations, or more generally Master equations (Gardiner, 1997; van Kampen, 1997), which describe the temporal evolution of the distribution of states in the population (Knight, 1972). Network models can be constructed by coupling the partial differential equations for each individual population (e.g, Brunel & Hakim, 1999; Omurtag et al., 2000; Nykamp & Tranchina, 2000). As stated above, DynamicLib is intended as a generic simulator of coupled systems of equations and an important aspect of this section is to show that DynamicLib can be employed to solve systems of coupled partial differential equations as well. But there is another aspect which relates to multi-level modelling: it can be shown that under suitable conditions, these systems of coupled Fokker-Planck equations must yield firing rates for populations that are given by Eq. 5. Indeed this must be the case, as it is claimed in the last section that the *steady state* of firing rates are approximated well by Eq. 5 and it is claimed in this section that the *dynamics* of large groups of LIF neurons is approximated well by population density techniques. Then, under the conditions for which Eq. 5 holds, they *must* describe the same steady state. We will demonstrate this explicitly for a network of one population (connected to an input population) and we will consider the radical difference in dynamics for a network which consists of two coupled populations when they are simulated by `OUAlgorithms` or by `PopulistAlgorithms`. This is an important demonstration of *extensibility*: a network which is described by `OUAlgorithms` can be changed by changing one line of code into a network described by `PopulistAlgorithms`.

We will first introduce the population density formalism and describe how it can be used in conjunction with DynamicLib. Since the development of the numerical algorithms to solve population density equations have led to a considerable body of code, this was moved into a dedicated library, PopulistLib.

Here we will consider neurons that are described by LIF dynamics:

$$\frac{dv}{dt} = -\gamma v + I,$$

below threshold potential $V_{threshold}$. When the membrane potential V passes $V_{threshold}$, the neuron emits a spike and its potential will be reset to V_{reset}. v is the rescaled membrane potential:

$$v = \frac{V - V_{reversal}}{V_{threshold} - V_{reversal}},$$

where $V_{reversal}$ is the neuron's reversal potential. I is any external contribution to the membrane potential, in this paper I is a stochastic input current.

Population density techniques describe infinitely large populations of neurons by means of a population density $\rho(v)$. $\rho(v)dv$ is defined as the fraction of neurons which have their membrane potential in the interval $[v, v + dv]$. It assumed that each neuron in the population receives a stochastic input, and that the distribution of this input is identical for all neurons. For the case where this input is a Poisson spike train and each spike causes a postsynaptic potential jump of magnitude h, the evolution of the density can be described by the following partial differential equation (see e.g. (Knight et al., 1996; Omurtag et al., 2000))

$$\begin{aligned} \frac{\partial \rho}{\partial t} - \gamma \frac{\partial}{\partial v}(\rho v) = {} & \sigma(t)\left\{\rho(v-h) - \rho(v)\right\} \\ & + r(t)\delta(v - v_{reset}). \end{aligned} \tag{11}$$

An important boundary condition is:

$$\rho(1, t) = 0, \tag{12}$$

which expresses the fact that $\rho(v, t) = 0$ for $v > 1$ and ensures that no unphysical probability can enter the system by leakage. Some neurons will be pushed across threshold ($v = 1$), by the input. Such neurons will spike and the membrane potential will be reset to v_{reset}. The flux across $v = 1$ is the

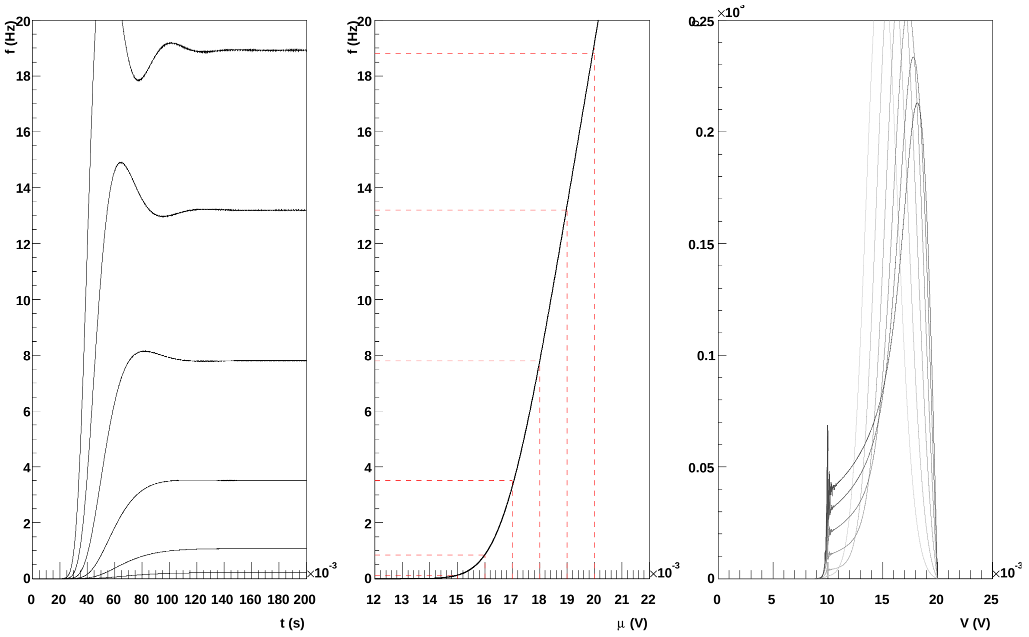


Fig. 10. This is a simulation for the same single population network that was shown in Fig. 8. The only difference in this figure is that the `PopulistAlgorithm` was used instead of the `OUAlgorithm`. The `PopulistAlgorithm` evolves a probability distribution function $\rho(v)$, which gives the distribution of membrane potentials in a neuronal population. From this, the firing rate can be calculated. As shown in the figure, the steady state rates are also well predicted by the $f - I$ curve, but the dynamics of the firing rate shows faster transient behaviour, which is confirmed by direct simulation of LIF neurons. This transient behaviour is not represented very well in Wilson-Cowan dynamics (see Fig. 8). The steady state distribution for $\rho(v)$ are shown on the right: the lightest curve corresponds to $\mu = 15$ mV, the darkest to $\mu = 20$ mV.

fraction of neurons per unit time that pass $v = 1$ and is therefore equal to the population firing rate. The firing rate is given by:

$$r(t) = \sigma(t) \int_{1-h}^{1} \rho(v', t) dv'. \tag{13}$$

The density that leaves the system due to synaptic input will be re-introduced at the reset potential, which is expressed by the second term on the right hand side of Eq. 11. This is the first `Algorithm` with a non trivial `NodeState`, namely the probability distribution function $\rho(v)$.

Several authors have published algorithms to solve Eq. 11 numerically (Omurtag et al., 2000), but to our knowledge none have been released as publicly available source code. PopulistLib contains algorithms developed by de Kamps (2003, 2006) and have been implemented in the form of a `PopulistAlgorithm`. In Fig. 10 we show the evolution of the state of a neuronal population which receives a fixed input rate over a single connection with fixed synaptic efficacy analogous to the situation described in the previous section. The code for this simulation is shown on `http://miind.sf.net/examples_nn_2008`. The code is nearly identical to that shown in Fig. 7, except that the `PopulistAlgorithm` requires more parameters, such as the number of bins used to represent the population density function $\rho(v)$.

In the simulation we model again a population receiving a stochastic input with given μ and σ. Like in the previous section, we increase μ from 15 mV to 20 mV, while σ is kept constant at 2 mV. For smaller μ, the firing rate converges quickly to its steady state which as in Fig. 8 is predicted by the $f - I$ curve. For higher μ, however, the firing rate starts to oscillate towards the steady state value. Although the differences between Fig. 8 and Fig. 10 look innocuous, for a two population network the behaviour between the two cases is rather different, as we shall see.

It is worthwhile to point out the similarities and differences between `OU_Algorithm` and `PopulistAlgorithm`. `OU_Algorithm` simulates a pseudo dynamics, which converges to sthe teady state firing rates, given by an analytic formula (Eq. 5) and are therefore based on what the firing rates *should be*. `PopulistAlgorithm` gives a statistical description of the dynamics of a large group of spiking LIF neurons, which is *exact* if the population is infinitely large (and already is a good approximation for several dozens of neurons). The output firing rates are not set by hand, but *emerge* as a consequence of the underlying neuronal dynamics. In particular the transient dynamics is described well.

5.1 Two Populations

Extending to larger networks is simple. As in previous cases, one merely adds more nodes and connections. Another possibility is to use an existing network. In Fig. 9 we show the simulation of a two population network consisting of an excitatory and an inhibitory population, which are connected to themselves and each other and which are driven by an external current. Changing just a single statement changes the simulation from a network where the neuronal dynamics is described by Wilson-Cowan-like dynamics into a network, which is described by population density techniques.

While in the single population case the differences were relatively minor (the simulations of Fig. 10 show just a slightly more oscillatory convergence to their steady states than the simulations of Fig. 8), the two population case shows a radically different behaviour. Where the two populations in Fig. 9 (which for convenience has been reproduced in Fig. 11) quickly converge to their steady state, the dynamics in Fig. 11 does not converge to a steady state at all, in-

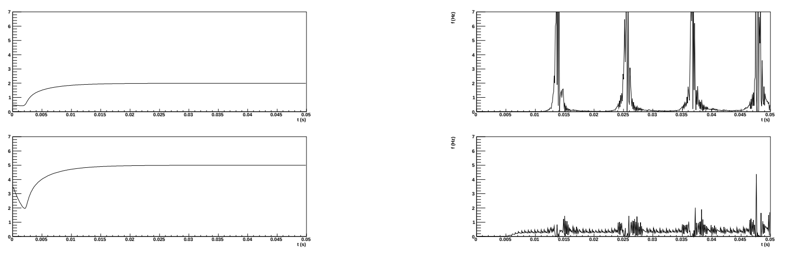


Fig. 11. The firing rate of a two population network. Network and neuronal parameters are identical to the simulation which used `OUAlgorithm`. The only difference with respect to that simulation is the neuronal dynamics. On the left hand side we have reproduced the simulations that used `OUAlgorithm`. On the right hand side, the corresponding firing rates for the network simulated with `PopulistAlgorithm`. Where for one population there was no radical difference between the two simulation methods, for this network there is.

stead displaying strong oscillatory behaviour. In observing the behaviour of the density of the populations, the behaviour can be explained. The inhibitory population has a smaller time constant and the population density will reach threshold before the excitatory population does. This means that the population starts to fire and since it is inhibitory, it tends to inhibit itself. This explains the high frequency oscillations which are present in both populations' firing rate. Once the excitatory population starts to fire, the excitatory population drives itself to higher firing rates. It will also drive the inhibitory population to a higher firing rate, but not quickly enough to stabilize its own firing rate. Both the inhibitory population and the excitatory population will be driven across threshold and the population density of both populations will end up around the reset potential. Then the whole process starts again. Such oscillatory behaviour is well documented for networks which have significant self connections (Mattia & Del Giudice, 2002; Sirovich et al., 2006).

Whether this simulation is a realistic description of the behaviour of neuronal populations remains to be seen. Refractory effects, spike latencies, intrinsic dynamics such as spike-frequency adaptation (Muller et al., 2007) and synaptic dynamics (Tsodyks, Pawelzik, & Makram, 1998) have been shown to have a moderating effect on this oscillatory behaviour and should be included in the simulation. At present, they are not. This is both a sobering and an amusing lesson: by merely changing a few lines of code one can not expect to turn a coarse level simulation into a realistic one without further understanding of what goes on at lower levels.

6 StructNetLib and ConnectionistLib

6.1 ConnectionistLib

An ANN can be represented with a `SparseImplementation`. In practice the only ANNs that feature in MIIND are layered feedforward networks. Although other networks could easily be created, we never needed them and we reiterate here that those who are looking for a wide variety of connectionist algorithms are better served elsewhere. Nevertheless, it is easy to create ANNs: `LayeredNetwork` is a feedforward ANN that relies on `SparseImplementation` for its implementation. What is necessary to turn a `SparseImplementation` into an ANN?

- Methods to *ReadIn* and *ReadOut* patterns to and from the network.
- An `Order` which describes the order in which the nodes are evaluated.
- A `SquashingFunction`

Exactly these attributes are provided with `LayeredNetwork`. Moreover a `BackpropTrainingAlgorithm` is provided, which is able to train the network so that it can produce the desired output in response to given input. An example of the use of `LayeredNetwork` is given on `http://miind.sf.net/examples_nn_2008`.

6.2 StructNetLib

While list of nodes (`NodeLinkCollection`) and architectures (e.g., `LayeredArchitecture`) are perfectly reasonable ways to create networks, sometimes one wants to determine the network structure by means of spatial relationships. Sometimes nodes should be connected if they are within a certain distance from each other, sometimes nodes have receptive fields that determine which nodes in a previous layer contribute to its input etc. It is therefore convenient to be able to describe networks in terms of spatial relations. For visualisation purposes it may also be important that the spatial structure of a network is represented accurately and not just by its connection structure. StructnetLib offers functionality to do just this.

A network can be defined in terms of `LinkRelations`. We illustrate the concept with an example. Ventral stream in visual cortex, for example, is often modelled by a hierarchical multilayer feedforward network, where the nodes in each layer have a receptive field: only nodes in a previous layer which are close enough to a given node will be input to this node. The structure looks deceptively simple, but in a multi-layered network of this kind there are many

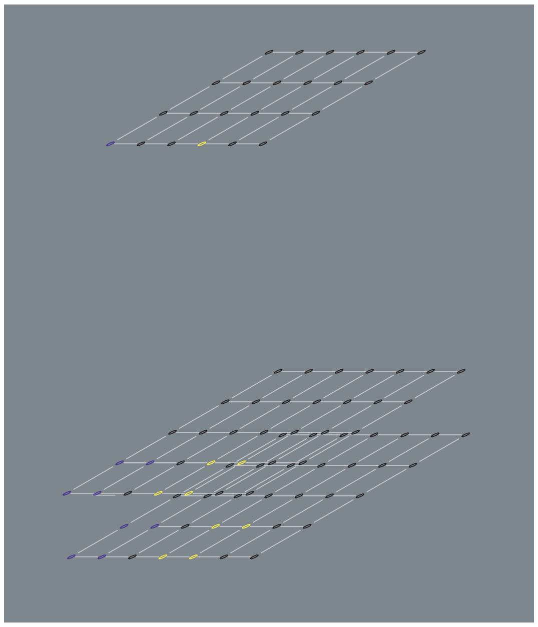


Fig. 12. Even a simple feedforward structure must be specified by a large number of parameters: the number of layers, the size of each layer in two dimensions, the size of the receptive field of a node in previous layers in two dimensions, the relative position of the receptive fields of two neighboring nodes. In the figure the lighter coloured nodes in the lower two layers are in the receptive field of the neuron in the higher layer. The connections are not shown. Recurrent structures of cortical areas involving multiple such networks need a great deal more parameters.

parameters, like the number of nodes in each layer in each dimension, the size of the receptive field, etc. It is tedious and very error-prone work to program these relations. It is an enormous boost to productivity if a user only needs to specify the parameters of a given spatial relation, rather than program it from scratch. MIIND provides the `DenseOverlapLinkRelation`, which is described in Fig. 12 and a number of other LinkRelations, which are a bit more specialized. It is easy to derive from `AbstractLinkRelation` if the existing `LinkRelation` types are not able to instantiate the desired architecture. They need to be programmed only once and if they are added to the framework, they will be available to other users. LinkRelations make it possible to define highly organized spatial structures which can be defined by a relatively small number of parameters and once they are defined, they can easily be reused. On `http://miind.sf.net/examples_nn_2008` we give code that creates a feed-forward network of the kind shown in Fig. 12 and show how it can be trained with backpropagation.

7 Visualization and storage of simulation results

7.1 Visualization of <u>DynamicLib</u> simulation results

As described above, `Handler` objects are responsible for collecting the simulation results and they can be used for visualization as well. The basic idea is that simulation results are written into a file to be analyzed at a later time. It is also possible to visualize a simulation whilst it is running. This direct form of visual feedback was instrumental in developing the `Populistalgorithm`. This visualization capability is derived by the `RootReportHandler`, which relies on the ROOT package. The ROOT package is a data storage and manipulation framework with powerful visualization capabilities. It is free, Open Source, and is created for efficient performance on high data volumes and is used by the new generation of high energy physics experiments in CERN.

An `AsciiReportHandler` writes out the simulation results in XML form. This can be instrumental in the debugging of new `Algorithms`. Other handlers compatible with MATLAB for example, could easily be created, but at the moment are not part of MIIND.

8 LayerMappingLib

8.1 Introduction

Most of the biologically inspired models of the ventral stream are hierarchical models. The term ”hierarchical models” describes a very broad class of models, which have in common that they possess nodes with activations coding for a certain feature at a certain position. The nodes are structured in layers. There are no interconnections between nodes within a layer, just connections from one layer to the next one in a feedforward fashion. The connections to a specific node are limited to a subset of the previous layer, the *receptive field*. The node is said to *pool* over the nodes in its receptive field. For the 2D case this principle is illustrated in Fig. 13.

An example of a hierarchical model for object recognition is HMAX (Riesenhuber & Poggio, 1999). Some node activations are determined by the maximum activation over the afferent nodes of the respective nodes. This is a non-linear operation and can not be implemented as a weighted sum with a squashing function. First let us consider networks where the activations are determined by

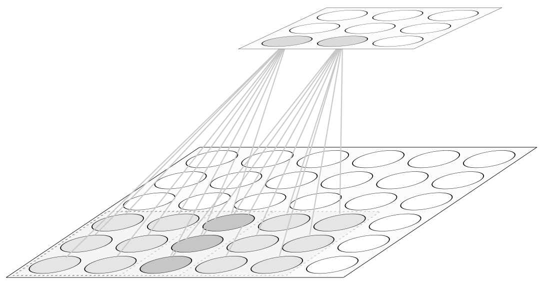


Fig. 13. A feedforward network with two-dimensionally spatial arranged nodes. The connections of the network are shown solely for the two nodes marked grey in the upper layer. Each node has a limited receptive field, which is in this case 3×3 in size. The receptive fields can overlap, depending on the so called skip size. The overlap is indicated by the darker nodes.

$$a_i = g(\sum_{a_j \in RF(a_i)} w_{ij} a_j), \tag{14}$$

where $RF(a_i)$ denotes the set of nodes that belong to the receptive field of node i and $g(x)$ is a squashing function. If the weight matrix $\mathbf{W}$ is shared by all nodes in a layer this leads to the idea of convolutional networks (LeCun et al., 1989). A layer in such a network represents an image. The activations in a layer are the result of Eq. 14. If the receptive field of each node in a particular layer consists of the afferent node and its surrounding nodes, then the layer can be seen as the result of a convolution with the weight matrix $\mathbf{W}$. Such convolutions are common in hierarchical models. They typically aim at extracting certain features. For example, in the first layer in HMAX filters are applied to the input image with different scales and orientations. Since a layer codes for a special feature we call it a *feature-map*.

This leads to the key idea of LayerMappingLib. A feature-map is represented by a `FeatureMapNode`. Hence the activations of such a feature-map-node correspond to a layer in a convolutional network, which is an array of nodes. `FeatureMapNodes` are connected in a `FeatureMapNetwork`. This allows us to describe a hierarchical model in terms of a `FeatureMapNetwork`. This facilitates the implementation of hierarchical models because the connections between `FeatureMapNodes` are fully specified by:

- the receptive field size, specifying the number of afferent nodes,
- the skip size, defining the overlap of receptive fields,
- a filter applied to each receptive field,
- and an *output skip size.*

The filters in LayerMappingLib are implemented as arbitrary functions. One of these functions is *Convolution()* which allows the implementation of linear filters. Non-linear filters such as the max-operation are implemented as special functions. An overview of the functions is given in the next section. A function

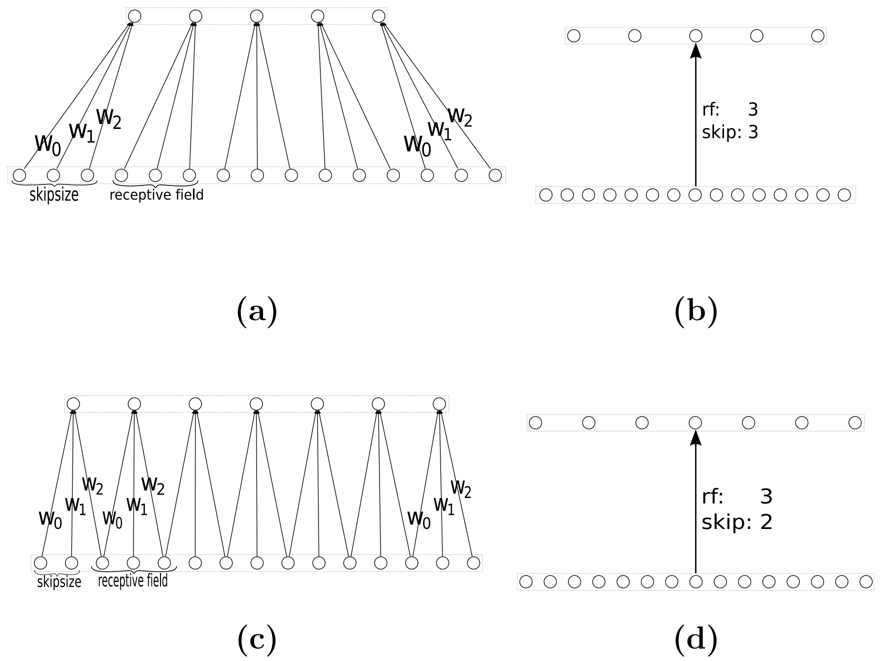


Fig. 14. (a) and (c) are examples of a *naïve* implementation of convolutional networks. All nodes in the upper layer share the same weights. The nodes in (a) and (c) have a receptive field size of 3, (a) has a skip size of 3 and (b) a skip size of 2. (b) and (d) are the representation of the same networks as a `FeatureMapNetwork`. Each network has two `FeatureMapNodes`. The filter used is a convolution with weight matrix (w_0, w_1, w_2)

operates on a receptive field and writes the result to one or more nodes in a feature-map. This explains why an output skip size is needed. If the node function writes to more than one node, the output skip size is greater than one. Consider for example the argmax function. There are two feature-maps, and the second is the result of the application of the argmax function to the first. In the second feature-map a node will have the same activation as the corresponding node in the first feature map if the corresponding node has the strongest activation in its receptive field in the first feature-map. If not, the activation is 0. Thus, argmax selects the strongest activation and suppresses the other ones.

8.2 Predefined Models

LayerMappingLib provides an infrastructure to realize hierarchical models. Although the implementation of a model can be based on LayerMappingLib, the description of the model in terms of C++ code still remains complex. As the library was implemented with the HMAX model in mind, HMAX can be instantiated as a predefined model. Predefined models can be instantiated with a single line of code. An example is given is the program shown in Fig.

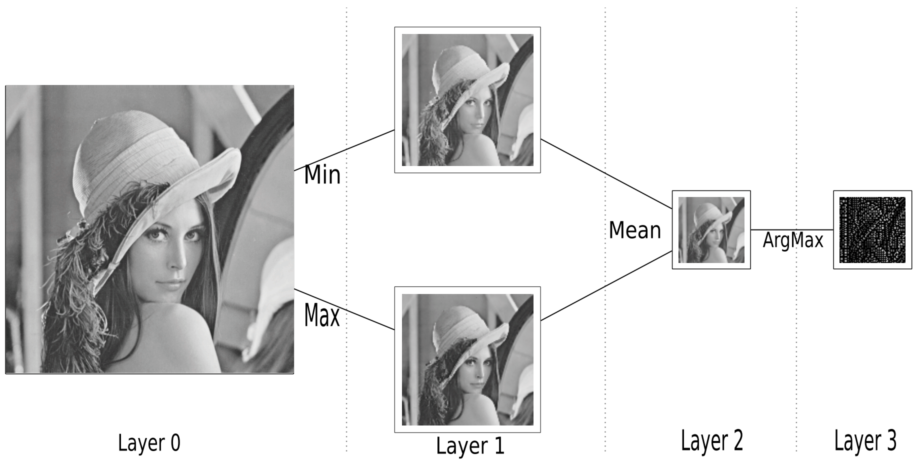


Fig. 15. Different filters are applied to an input image yielding multiple feature-maps. The feature-map in layer 0 is the input image. The feature-maps in layer 1 are obtained by applying a min (respectively max) filter to the input image. The receptive field is 2×2 and the skip size is 2×2, too. Thus a sub-sampling happened, where the receptive fields are not overlapping and the size of the resulting feature-maps is halved. The feature-map in layer 2 results from a mean filter on the feature-maps in layer 1, demonstrating the ability to combine different feature-maps. Again skip size and receptive field size are 2×2. The feature-map in the third layer is the result of the `argmax` filter. `argmax` suppresses all but the maximum activation in the receptive field. This implements a winner-take-all mechanism. Here the receptive field and skip size are again 2×2, but in contrast to the previous layers the output skip size is 2×2. Thus the size of the feature-map equals its predecessor.

```
#include <LayerMappingLib/LayerMappingLib.h>

using namespace LayerMappingLib;

int main( int argc, char** argv )
{
  network myModel = Models::SimpleTest( 16, 16 );

  FeatureMap<double> input_layer =
        myModel.input_activation().front();
  generate( input_layer.begin(),
            input_layer.end(),
            rand );
  evolve( myModel.begin(),
          myModel.end() );

  myModel.debug_print();
}
```

Fig. 16. SimpleTest, a brief example program.

16.

The network instantiated by calling *SimpleTest* in line 7 is sketched in Fig. 15. We have chosen 16×16 as the dimension of the input layer. In line 11 random noise is assigned to the input feature-map. The input feature-map can be any other pattern, typically an image. The network is evolved by the function *evolve* in line 14. *debug_print* prints the activation of the network to the console. To inspect larger networks *debug_print* gets rather useless, for this purpose the graphical user interface coming with LayerMappingLib is more appropriate.

There are different usage scenarios for LayerMappingLib. The simplest is to assign an image to the input layer and evolve it to get a feature vector as a response of the model. This feature vector could be used, for example, in a further classification step.

Another way is to extend an existing model or create a new model. A variety of predefined filters are therfore implemented and supplied with LayerMappingLib. These filters can be applied to feature-maps. The currently available filters are:

- *Min*, *Max*, *Mean*
- *Sum*, *Product*
- *ArgMax*
- *Convolution*
- *Perceptron*

Here *Perceptron* is a function that applies a convolution to a weight matrix, adds a bias and applies a squashing function to the given input. The *Perceptron* function already provides all necessary functionality to implement convolutional networks, as proposed by LeCun, Bottou, Bengio, and Haffner (1998). However, the intention of LayerMappingLib is to evolve the model by successive application of filter matrices, not to learn the parameters. Parameter learning is delegated to ConnectionismLib , the part of MIIND dedicated to connectionist learning algorithms.

Convolution is a very generic function. Depending on its filter matrix it can be used for different purposes. In the implementation of HMAX it was used for orientation filtering with second derivative gaussian filters.

The typical steps for building a new model are as follows:

- Create a `FeatureMapNetwork`
- Specify the skip size, receptive field size, the filter and the predecessors of a `FeatureMapNode` and add it to the network. Often feature-maps in a layer have the same parameters, so all `FeatureMapNodes` in a layer can be instantiated with the same parameters.
- Repeat the previous step until the network has the desired structure.
- *evolve* the network and read the data needed for further processing from the respective feature-maps.

If the filters of LayerMappingLib are not sufficient to implement a model, custom filters can be added. The library was designed to allow the extension of the filter collection in a convenient way. This is done by using the factory pattern. A new filter simply must be registered to a factory class. The structure of the models presented so far is strictly feedforward. To implement inhibition and recurrence, networks can be subsumed in a network ensemble. In an ensemble connections between networks are allowed. This allows to implement a disinhibition mechanism as proposed by (van der Velde et al., 2004) with only little effort. Clearly HMAX is not the only model of interest, but it serves as a good illustration of how filter banks can be combined into networks. We refer to `http://miind.sourceforge.net/apiDocs/miind_LayerMappingLib/html/index.html` for a detailed explanation of how SimpleTest is created.

9 Discussion

In this paper we have demonstrated the various components of MIIND. SparseImplementationLib can be used on its own. It can be applied wherever sparse irregular networks must be modelled and is not restricted to neuronal networks. DynamicLib is a framework for simulating network processes. We have given demonstrations on how it can be used. It can also be adopted for other processes and need not be restricted to neuronal processes only. By introducing suitable algorithms of one's own design, one can model essentially every process at the nodes. By suitably choosing the type of the connections, one can model many types of networks.

In the introduction, we mentioned the concepts of extensibility and detailing. Because dynamical models are instantiated in the same way, parts of an earlier model can easily be inserted into a more complex model: essentially one has to provide the appropriate calls to *AddNode* and *MakeFirstInputOfSecond* as subroutines. These routines then correspond to sub modules of the network. We have given an explicit demonstration of *detailing*: a network that initially was created with Wilson-Cowan dynamics can easily be transformed into one which uses population density techniques, which gives a much more realistic description of neuronal dynamics. We could take the concept of detailing even further by encapsulating one of the existing neuronal simulators, such as e.g. NEURON, GENESIS or NEST and encapsulating such simulators in an `Algorithm` interface. Each node represents a population with full (or a least non-sparse) connectivity. The basic assumption is that interactions between populations can be modelled using `DynamicNetwork`: connections between individual neurons in the two populations can not be represented and the influence of one population on another must somehow be described statistically, so that it can be represented by a single network connection. If this assumption holds very large networks can be simulated in great detail. We believe that this assumption will hold quite generally. This is the essential application area of DynamicLib: sparsely connected networks, but nodes where in principle anything can happen. Another way to use detailing is first to create a network where the coarse structure of a neuronal area is known and to perform simulations. Later, if more is known about the detailed connectivity of the area, extra connections can be added by calling *MakeFirstInputOfSecond* again. So networks can be continually refined.

9.1 MIIND in relation with other neural simulators

Originally, MIIND was developed for our own use. It turned out to be worthwhile to isolate C++ classes for reuse (in our own models) and also to make the code more accessible to other users. The factorization of MIIND into small units with a clear specialization was the first attempt to make MIIND attractive for other C++ developers. Recently, the issue of interoperability has gained momentum (Cannon et al., 2007). For a large group of modellers, Python has become the *de facto* standard and the development of PyNN (Brette et al., 2007) is an interesting attempt to provide a common user interface for neural simulators. MIIND will certainly need to develop a Python interface.

The question of whether a sensible definition within PyNN is possible is an interesting one. By its nature, with its emphasis on Wilson-Cowan dynamics and population density methods, it is one level above the most widely used spiking neuron simulators. Most simulation concepts of NEST, GENESIS, MOOSE, NEURON and others do not translate directly. To include MIIND in PyNN, the *AddNode* and *MakeFirstConnectionOfSecond* would need corresponding calls in PyNN at the level of populations. MIIND may develop into a framework that can drive simulations of spiking neurons, but then needs parallelization (see section 9.2) to be efficient. It would then allow direct comparisons between population density methods and spiking neuron simulations.

Although MIIND can represent ANNs efficiently and contains a number of connectionist algorithms, Lens (`http://tedlab.mit.edu/~dr/Lens`), PDP++(O'Reilly & Rudy, 2001) or its successor Emergent (`http://grey.colorado.edu/emergent/index.php/Main_Page`) and NSL (`http://www.neuralsimulationlanguage.org/` offer more algorithms for connectionist modelling. MIIND distinguishes itself from these packages with its emphasis on high level algorithms for neural dynamics. To our knowledge none of these packages offer Wilson-Cowan dynamics or population density methods.

LayerMappingLib shares some concepts with Topographica (`http://topographica.org`). The layers correspond to Topographica's *sheets* and similar filters have been implemented. LayerMappingLib does not concern itself with the development of a map, however. It represents rectangular two-dimensional maps and implements filter operations between them. It exploits the rectangular geometry of the map to implement a efficient implementation of large filter bank structures, which are present in models such as HMAX and successors (Riesenhuber & Poggio, 1999; Serre et al., 2007).

Currently, MIIND is used for high level models of visual attention and the so-called neural blackboard architecture (van der Velde & de Kamps, 2001; de Kamps & van der Velde, 2001; van der Velde, van der Voort van der Kleij, & de Kamps, 2004; van der Velde & de Kamps, 2006; de Kamps & van der Velde, 2008). In terms of modelling this entails interacting hierarchies of neural networks and the simulation of local neural circuits (van der Velde & de Kamps, 2001; de Kamps, 2005; van der Velde & de Kamps, 2006), usually in terms of rate models. Although there is a large number of models in cognitive neuroscience that all use the same, or very similar techniques, (see e.g., O'Reilly & Rudy, 2001; Lanyon & Denham, 2004; Usher & Niebur, 1996; Hamker, 2005; O'Reilly, 2006), Emergent (PDP++) is the only other software package that we are aware of at this level. Most models seem to be created from scratch and hence the situation in cognitive neuroscience is notably different from that in computational neuroscience. Here no one would consider to start a model from scratch, given the neural simulators that are already available. Emergent is a package which is very successful and has been used in many high level models of cognition (e.g., O'Reilly & Rudy, 2001; O'Reilly, 2006). It imposes a rather specific modelling approach, however, and it is not always clear to us how we could reimplement our own models in Emergent. It also does not seem to offer algorithms to simulate neural dynamics at the population level.

We believe that MIIND is rather unique in the way that its low level functionality is factored out. Nearly every simulator that deals with biologically realistic networks has had to deal with the issue of'representing sparse networks. Most people, however, have not considered this to be a problem in its own right, one which deserves a good generic implementation, or at least a discussion in terms of Design Patterns (Gamma et al., 1994) so that others may profit from the experience that implementing a solution brings. Likewise, the creation of a central simulation loop needs to be solved in every simulator. It entails not only driving the simulator, but storing the intermediate results and writing simulation conditions to log files. If this is all properly taken into account, this mundane problem is not so mundane anymore. And although nearly every simulation package has had to solve this problem, the way in which it is solved is usually hidden deep in the source code. We try to make these low level solutions accessible so that at the very least, they are open for discussion, but ideally they will be absorbed into the code of others.

We would encourage other developers to do the same. In order to create packages, such as the ones discussed in (Brette et al., 2007), their developers have had to solve a large number of low level problems which might find application outside the specific simulation context for which they were created. Such low level solutions would be the core of Design Patterns for computational and cognitive neuroscience. To encourage reuse and validation of our models, we will upload our models in ModelDB (`http://senselab.med.yale.edu/modeldb`).

9.2 Future Challenges

We are now able to configure simulate networks of populations with LIF dynamics, both in the super- and the supra-threshold regime (Brunel, 2000). While it has been demonstrated that population density methods describe the response of LIF neurons well, it has not been demonstrated that LIF neurons describe biological neurons sufficiently adequately. Approaches which go beyond LIF neurons are (e.g,. Brette & Gerstner, 2005; Apfaltrer et al., 2006; Muller et al., 2007). At present, they have not been implemented in publicly available source code, but the separation between neuronal dynamics and spike statistics described by de Kamps (2003, 2006) make the algorithms described there suitable for such extensions. Since LIF neurons may fall short of the biological realism that is required, developing such algorithms is an important challenge. We expect to be finished with providing Python interfaces by the time this publication appears.

`DynamicNetwork` is an obvious candidate for parallelization. A `DynamicNetwork` based on a parallel implementation could be used to drive spiking neuron simulators. We would like to retain the external interface of `DynamicNetwork` but provide a new implementation of `DynamicNetworkImplementation` based on MPI (Gropp, Lusk, & Skjellum, 1994). In the longer run the algorithm of Morrison, Mehring, Geisel, Aertsen, and Diesmann (2005) seems well suited for MIIND's network model and is perhaps the better choice.

Acknowledgement

We are grateful to two anonymous reviewers whose comments have improved the original document considerably.

References

Amit, D. J., & Brunel, N. (1997a). Dynamics of a recurrent network of spiking neurons before and following learning. *Network: Computation in Neural Systems*, *8*, 123-152.

Amit, D. J., & Brunel, N. (1997b). Model of global spontaneous activity and local structured activity during delay periods in the cerebral cortex. *Cerebral Cortex*, *7*, 237-252.

Amit, D. J., & Tsodyks, M. V. (1991). Quantitative study of attractor neural network retrieving at low spike rates: I. substrate - spikes, rates and neuronal gain. *Network*, *2*, 259-273.

Apfaltrer, F., Ly, C., & Tranchina, D. (2006). Population density methods for stochastic neurons with realistic synaptic kinetics:. *Network: Computation in Neural Systems*, *17*, 373-418.

Brette, R., & Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronalactivity. *Journal of Physiology*, *94*, 3637-3642.

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Jr., F. C. H., Zirpe, M., Natschlager, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Davison, E. M. A. P., Boustani, S. E., & Destexhe, A. (2007). Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, *23(3)*, 349-398.

Brooks, R. A. (1991). Intelligence without reason. In J. Myopoulos & R. Reiter (Eds.), *Proceedings of the 12th international joint conference on artificial intelligence (IJCAI-91)* (p. 569-595). Sydney, Australia: Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *Journal of Computational Neuroscience*, *8*, 183-208.

Brunel, N., & Hakim, V. (1999). Fast global oscillations in networks of integrate-and-fire neurons with low firing rates. *Neural Computation*, *11*, 1621-1671.

la Camera, G., Rauch, A., Lscher, H. R., Senn, W., & Fusi, S. (2004). Minimal models of adapted neuronal response to in vivo-like input currents. *Neural Computation*, *16(10)*, 2101-2124.

Cannon, R. C., gewaltig, M.-O., Gleeson, P., Bhalla, U. S., Cornelis, H., Hines, M. L., Howell, F. W., Muller, E., Stiles, J. R., Wills, S., & de Schutter, E. (2007). Interoperability of neuroscience modeling software: Current status and future directions. *Neuroinformatics*, *5*, 127-138.

Djurfeldt, M., Lundqvist, M., Johansson, C., Rehn, M., Ekeberg, Ö.., & Lansner, A. (2008). Brain-scale simulation of the neocortex on the blue gene/l supercomputer. *IBM Journal of Research and Development*, *52*, 31-42.

Fromherz, P. (2003). Semiconductor chips with ion channels, nerve cells and brain. *Physica E*, *16*, 24-34.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, *4*, 193-202.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns - elements of reusable object-oriented software.* Reading, MA: Addison-Wesley.

Gardiner, C. W. (1997). *Handbook of stochastic methods for physics, chemistry and the natural sciences.* New York Berlin Heidelberg Tokyo: Springer-Verlag.

Gerstner, W. (1995). Time structure of the activity in neural network models. *Physical Review E*, *51*, 738-758.

Gerstner, W., & Kistler, W. (2002). *Spiking neuron models - single neurons, populations, plasticity.* Cambridge: Cambridge University Press.

Gropp, W., Lusk, E., & Skjellum, A. (1994). *Using MPI - portable parallel programming with the message-passing interface.* Cambridge MA, London England: The MIT Press.

Hamker, F. H. (2005). The reentry hypothesis: The putative interaction of the frontal eye field, ventrolateral prefrontal cortex, and areas V4, IT for attention and eye movement. *Cerebral Cortex*, *15*, 431-447.

Haskell, E., Nykamp, D. Q., & Tranchina, D. (2001). Population density methods for large-scale modelling of neuronal networks with realistic synaptic kinetics: cutting the dimension down to size. *Network: Computation in Neural Systems*, *12*, 141-174.

van Kampen, N. G. (1997). *Stochastic processes in physics and chemistry.* Amsterdam: North-Holland.

de Kamps, M. (2003). A simple and stable numerical solution for the population density equation. *Neural Computation*, *15*, 2129-2146.

de Kamps, M. (2005). A model for delay activity without recurrent excitation. *Lecture Notes in Computer Science*, *3696*, 229-234.

de Kamps, M. (2006). An analytic solution of the reentrant poisson master equation and its application in the simulation of large groups of spiking neurons. In *Proceedings WCCI2006 (IJCNN2006).* Vancouver, CA.

de Kamps, M., & Baier, V. (2007). Multiple interacting instantiations of neuronal dynamics (miind): a library for rapid prototyping of models in cognitive neuroscience. In *Proceedings IJCNN2007.* Florida, USA.

de Kamps, M., & van der Velde, F. (2001). From artificial neural networks to spiking populations of neurons and back again. *Neural Networks*, *14*, 941-953.

de Kamps, M., & van der Velde, F. (2008). A neurodynamic model for pop-out. *In Preparation.*

Knight, B. W. (1972). Dynamics of encoding in a population of neurons. *Journal of general Physiology*, *59*, 734-766.

Knight, B. W., Manin, D., & Sirovich, L. (1996). Dynamical models of interacting neuron populations in visual cortex. In E. C. Gerf (Ed.), *Symposium on robotics and cybernetics: Computational engineering in systems applications.* France: Cite Scientifique.

Koenig, A., & Moo, B. E. (2005). Templates and duck typing. *Dr. Dobbs Portal.*

Lanyon, L. J., & Denham, S. L. (2004). A model of active visual search with object-based attention guiding scan paths. *Neural Networks*, *17*, 873-897.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86(11)*, 2278-2324.

LeCun, Y., Jackel, L. D., Boser, B., Denker, J. S., Graf, H. P., Guyon, I., Henderson, D., Howard, R. E., & Hubbard, W. (1989). Handwritten digit recognition: Applications of neural net chips and automatic learning. *IEEE Communication*, 41-46. (invited paper)

Markram, H. (2006). The blue brain project. *Nature Reviews Neuroscience*, *7*(2), 153–160.

Mattia, M., & Del Giudice, P. (2002). Population dynamics of interacting spiking neurons. *Phys. Rev. E*, *66*(5), 051917.

Morrison, A., Aertsen, A., & Diesmann, M. (2007). Spike-timing-dependent plasticity in balanced random networks. *Neural Computation*, *19*, 1437-1467.

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., & Diesmann, M. (2005). Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Computation*, *17*, 1776-1801.

Muller, E., Buesing, L., Schemmel, J., & Meier, K. (2007). Spike-frequency adapting neural ensembles: Beyond mean adaptation and renewal theories. *Neural Computation*, *19*(11), 2958-3010.

Navarro, X., Krueger, T. B., Lago, N., Micera, S., Stieglitz, T., & Dario, P. (2005). A critical review of interfaces with the peripheral nervous system for the control of neuroprostheses and hybrid bionic systems. *Journal of the Peripheral Nervous System*, *10(3)*, 229-258.

Nicolelis, M. A., Dimitrov, D., Carmena, J. M., Crist, R., Lehew, G., Kralik, J. D., & Wise, S. P. (2003). Chronic, multisite, multielectrode recordings in macaque monkeys. *Proceedings of the National Academy of Sciences of the United States of America*, *100(19)*, 11041-11046.

Nykamp, D. Q., & Tranchina, D. (2000). A population density approach that facilitates large-scale modeling of neural networks: Analysis and an application to orientation tuning. *Journal of Computational Neuroscience*, *8*, 19-50.

Omurtag, A., Knight, B. W., & Sirovich, L. (2000). On the simulation of large populations of neurons. *Journal of Computational Neuroscience*, *8*, 51-63.

O'Reilly, R. (2006). Biologically based computational models of high-level cognition. *Science*, *314*, 91-94.

O'Reilly, R., & Rudy, J. W. (2001). Conjunctive representations in learning and memory. principles of learning in the neocortex and hippocampus. *Psychological Review*, *108*, 311-345.

Pfeifer, R., & Scheier, C. (1999). *Understanding intelligence.* MIT Press.

Prechelt, L. (2000). An empirical comparison of seven programming languages.

Computer, *33*(10), 23-29.

Renart, A., Brunel, N., & Wang, X. (2004). Mean-field theory of recurrent cortical networks: From irregular spiking neurons to working memory. In J. Feng (Ed.), *Computational neuroscience : A comprehensive approach.* Boca Raton: CRC Press.

Ricciardi, L. M. (1977). *Diffusion processes and related topics in biology.* Berlin: Springer Verlag.

Riesenhuber, M., & Poggio, T. (1999). Hierarchical models of object recognition in cortex. *Nature Neuroscience*, *2*, 1019-1025.

Schoen, I., & Fromherz, P. (2007). The mechanism of extracellular stimulation of nerve cells on an electrolyte-oxide-semiconductor capacitor. *Biophysics Journal*, *92*, 1096-1111.

Serre, T., Wolf, L., Bileschi, S., Riesenhuber, M., & Poggio, T. (2007). Object recognition with cortex-like mechanisms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *29 (3)*, 411-426.

Sirovich, L., Omurtag, A., & Lubliner, K. (2006). Dynamics of neural populations: Stability and synchrony. *Network: Computation in neural systems*, *17*, 249-272.

Tsodyks, M. V., Pawelzik, K., & Makram, H. (1998). Neural networks with dynamic synapses. *Neural Computation*, *10*, 821-835.

Usher, M., & Niebur, E. (1996). Modeling the temporal dynamics of it neurons in visual search: A mechanism for selective top-down attention. *Journal of Cognitive Neuroscience*, *8*, 311-327.

van der Velde, F., & de Kamps, M. (2001). From knowing what to knowing where: Modeling object-based attention with feedback disinhibition of activation. *Journal of Cognitive Neuroscience*, *13(4)*, 479-491.

van der Velde, F., & de Kamps, M. (2006). Neural blackboard architectures of combinatorial structures in cognition. *Behavioral and Brain Sciences*, *29(1)*, 37-70.

van der Velde, F., de Kamps, M., & van der Voort van der Kleij, G. (2004). Closed-loop attention model for visual search. *Neurocomputing*, *58-60*, 607-612.

van der Velde, F., van der Voort van der Kleij, G., & de Kamps, M. (2004). Increasing number of objects impairs binding in visual working memory. *Neurocomputing*, *58-60*, 599-605.

Webb, B. (2001). Can robots make good models of biological behaviour? *Behavioral and Brain Sciences*, *24*, 1033-1050.

Wilson, H. R., & Cowan, J. D. (1972). Excitatory and inhibitory interactions in localized populations of model neurons. *Biophysical Journal*, *12*, 1–23.