Generating Neural Network Connectivity from a Visual Representation

Schriftliche Prüfungsarbeit
für die Master-Prüfung des Studiengangs Angewandte Informatik
an der Ruhr-Universität Bochum

vorgelegt von

Herbers, Patrick
108012219163

09.11.2017

Prof. Dr. Sen Cheng
M.Sc. Eng. Sandra Diaz Pier

Contents

Chapter 1

Introduction

The brain is an incredibly complex organ, and research on it has been done for over a century. Neuroscience is a relatively young field, and it aims to understand the function and complex structure of the brain. The brain's complexity stems mostly from the interconnection of its cells. There are 10^{12} cells in the human brain, and up to 10^{15} connections between them [18]. To unravel the mystery of the brain, we have to understand its inner workings. In science, proving a hypothesis is usually done by measuring, however measuring the neuronal activity inside the brain turns out to be a difficult task. Because of this, neuroscientists have to resort to simulating parts of the brain using modern computing power. Thus, network simulation software has become an integral part of modern neuroscience.

Writing the code for these simulations can be time consuming and requires expert knowledge in computer science. Network simulation software, such as *NEST* [20] or *NEURON* [17], can offer simulations of these networks at a high performance. There is a multitude of different simulators in use, with varying scope and functionality. Typically, to operate a simulator, the user needs in-depth technical knowledge for even basic tasks. Neuroscience has become an extremely interdisciplinary field where biology, psychology, medicine, and computer science intersect. Separating the technical knowledge from the scientific knowledge is an important step in making neuroscience more accessible for scientists of all fields. One way to achieve this separation is by guiding the user with a graphical user interface (GUI). A well designed GUI can teach a user its features in an intuitive way, and reduces mental load when designing network models.

Another advantage of a separation of simulation and design is the independence of the simulation software used. If a model can be simulated with different simulators without reimplementation, additional methods for cross validation of the results is easily available. Updating simulation software would not break model definitions, and switching

the simulation software would be trivial. To separate design from simulation, a translation between the two domains has to be performed.

1.1 Network Models

When trying to analyze the brain, network models are an important tool to get an insight of its inner workings. These models are mathematical abstractions which can be simulated computationally. These computations are usually done by neuronal simulators, which offer various tools for creating and simulating models, and can optimize the performance of the simulation. The output of a simulated model can be compared to biological measurements to test its validity.

The computational models are simplified versions of the brain's biological components. A computational model includes abstractions to increase performance. This section will introduce the neuroscientific terms and mathematical abstractions that are relevant to this work. A model usually consists of **neurons**, **synapses** and **inputs**. A **neuron** is a cell made up of three parts: the *dendrites*, the *soma*, and the *axon* [11]. Signals from other neurons are received by the dendrites, and passed along a tree-like structure (*dendritic tree*) to the soma. The soma is the central part of the neuron; when the incoming signals pass a certain threshold, the neuron fires. The axon passes the outgoing signal to other neurons. These axon-dendrite connections are called **synapses**. A synapse's sending neuron is called the **presynaptic neuron**, and the receiving neuron the **postsynaptic neuron**. The signals a neuron fires are short electrical pulses, also called **spikes** or action potentials. These signals are transmitted (usually through a biochemical process) through the synapse to the postsynaptic neuron's cell membrane.

To increase the simulation performance, the different processes of neurons and synapses are abstracted through computational models. In simulation software, neurons are often simplified as a point neuron model, which simulates the interactions of the soma, axon, and dendrites in the form of mathematical formulas. Synapses are also abstracted to fit mathematical models, generally controlled by a **synaptic weight** (synaptic strength) or and **synaptic delay**. The synaptic weight indicates the influence a spike has on its postsynaptic neuron. The synaptic delay corresponds to the time a spike has to travel to reach the postsynaptic neuron. All synapses can be split into two types: **excitatory** and **inhibitory** synapses. Excitatory synapses are defined by a positive synaptic weight and increase the membrane potential of the postsynaptic neuron. Inhibitory synapses have a negative synaptic weight and decrease the membrane potential. Some synapse models also support changing synaptic weights, called **synaptic plasticity**. Plasticity is an important aspect of the network's learning process. The underlying plasticity models

are more complex than static synapses, take more computational power to simulate, and have a larger parameter space.

Since models of the brain can become quite complex with a large number of neurons, the neurons can be grouped together into **neuron populations** when designing a network. The neurons are typically grouped to organize anatomical or functional structures together. Similarly, the connections between the neurons in the neuron populations can be grouped into **projections**. Projections are also directional and have a source and target population. A projection encompasses all the connections between neurons of the source population and the neurons of the target population. This can either be a simple grouping of connection instances, or an abstract pattern. A connection pattern can generate the actual connection instances of a projection when the simulator is building the model. This abstraction can save memory and workload when designing the model. Instead of defining millions of connections by hand, a projection can build them based on the pattern's algorithm.

To take into account the influence from regions not explicitly defined in the model, network models usually have some form of **input** to their populations. These inputs can be sensory signals, simplified areas of the brain that are not in the model, or measured data that is fed into the simulation. The mechanism of the input can vary, from applying current directly to a neuron's cell membrane, to generating random spikes for a given duration. Usually, the target of the input is a population in the model, but single neurons can also be stimulated.

Combining populations, projections and inputs into a model allows for recreating many of the neuronal structures that can be found in the brain, given the synapse and neuron models offer an accurate recreation of the functionality of the cells. Current simulations lack the capacity to model the complexity of the brain. The more connections and neurons a model has, the higher the computational power needed for simulation. These limitations can be caused by overhead in distributed systems and memory consumption [16]. Projects like the Blue Brain Project [21] aim to simulate the whole human brain using modern supercomputers. Current supercomputers are able to simulate models with up to 10^8 cells. Research focuses on simulating simplified versions of the brain with a percentage of the neurons and connections, or simulating a specific subset or area of the brain.

1.2 Problem Statement

When a neuroscientist wants to test a new hypothesis with a modeling approach, he first has to design the model. Usually, this is done analogue on paper. The populations and

their connections of the model have to be designed by hand, and the way of visualizing the model has to be reinvented for every model. This way of visualizing introduces ambiguity as there is no concrete standard for network models. After a model has been designed by hand, it has to be tested in a simulator. Translating the designed model into a simulation specific description can be a daunting task for the user, and requires in-depth knowledge of the simulator. Even when a model has been implemented in a simulation language, understanding the simulation code is often difficult [4], and transferring the code to a different simulator is complicated, as various simulators offer completely different interfaces for user interaction. This project attempts to find a common visual language for representing connectivity, while also allowing automatic translation of this visual language to any simulator.

In order to reduce the complexity of the interactions between user and simulator, a GUI is proposed which can display the model information using a visual representation. The GUI can be connected to a translator in order to produce simulator independent connectivity data from a created model. The translator can then enable a simulation using different neuronal simulators. This allows a user to simulate a model without extensive knowledge of the underlying simulator.

The visual representation that has to be developed needs to be intuitive for researchers of neuroscience and to convey large amounts of information quickly to the user. The focus of the representation is on the connectivity between groups of neurons, also called the *connectome* [28] of a model. Due to the large amounts of data in a connectome, visualizing every synaptic connection is not feasible. Instead, the visualization has to focus on the projections as template groups of connections, using connectivity patterns. The visualization has to balance abstracting large amounts of data and giving detailed information, while providing deep insight into the model. Information to visualize can include the number of connections, the pattern of connectivity, or the type of computational model.

Development of the GUI itself is **not** a part of this project. Instead, the GUI is developed in parallel by a team of the *Grupo de Modelado y Realidad Virtual (GMRV)* at the *Universidad Rey Juan Carlos (URJC)* in Madrid. The GMRV is responsible for implementing the visual language developed in this project in their visualization framework *NeuroScheme* (see Section 1.3.4). The visual language was developed to fit and complement the framework. To aid the creation of the GUI, conducting user interviews and finding user requirements was one of the tasks for this project.

The translator poses the link between the GUI and the simulator. The translation requires the visual representation to be bound to data structures which can be passed from the GUI to the translator. The translator has to be developed with a modular

design in mind to allow simulator independence and to ensure compatibility with the in parallel developed version of NeuroScheme.

1.3 State of Technology

Designing a visual language and translating it requires various technologies and software. To solve the tasks of this project, the following solutions have to be found:

- A GUI for displaying the visual language.
- A standard for defining network models.
- A way of expressing and generating connectivity in a simulator independent way.
- Simulators for the network simulation.
- An interface to the simulators.

This chapter will introduce the software and specifications used in this project, and show related work that has been done on visualization.

1.3.1 The Connection Set Algebra

When describing the connectivity of a model, it is often done without adherence to standards [23]. Implementing the connectivity in a simulator can be difficult and is prone to ambiguity in the model description. Model definitions either rely on complex written out descriptions of their connectivity functions, pieces of code written for a specific simulator, or connectivity patterns without formal definitions. This makes the reproducibility of network models difficult, as models are designed for a specific simulation environment. Transferring the code for generating the connectivity makes reproducing the experiment easier, but implementing a model with complex network structures in a different simulator requires just as much work as building from scratch. To provide a formal standard that can be used to convey the connectivity of a model, not only in written text and formulas, but also among neuronal simulators, the Connection Set Algebra (CSA) was developed by Mikael Djurfeldt [7].

The Connection Set Algebra is a mathematical representation of connections between populations of neurons. CSA is a way to formalize connectivity patterns in an abstract language using set algebra. With this abstract formalism a connectivity pattern becomes independent of the different pattern definitions of various simulators. This independence is an important aspect of the modular nature of CSA. The abstract connectivity formalism that is offered by the algebra allows linking CSA with any simulator.

To formalize the connectivity between two neuron populations, the connection set algebra introduces the concept of a connection set. A connection set contains the information needed for a projection, including which neurons are connected, and parameter values for the synaptic model. The actual connectivity of the connection set is described in the connection set's mask. A mask $\bar{M}$ can be interpreted in three different ways: In traditional neuroscience terms, it can be seen as a connection matrix between two populations, where a boolean entry $\bar{M}_{ij}$ indicates a connection between the presynaptic neuron with index i and the postsynaptic neuron with index j. Figure 1.1a and Figure 1.1b show how connectivity can be represented by a connection matrix. This is similar to the interpretation as a mapping function $\bar{M} : \mathbb{N}_0 \times \mathbb{N}_0 \to \{\mathcal{F}, \mathcal{T}\}$ that maps a boolean value to a tuple of indices. For use in the connection set algebra, the mask is also seen as a set of tuples. A tuple (i, j) is part of the set if there exists a connection between the neurons. This interpretation allows the Connection Set Algebra to use set arithmetic to create new sets.

An important part for the simulator independence of CSA is that the masks are infinite in nature. An infinite mask is independent of the size of the populations it connects. Because the mask does not need any information about the neurons it connects, multiple populations can be connected with the same expression, and population size can be changed without having to change the connectivity.

The masks can be created from elementary masks provided by the algebra, which serve as a base to all other masks. Elementary masks try to cover the most basic cases of connectivity patterns. The elementary masks are as follows:

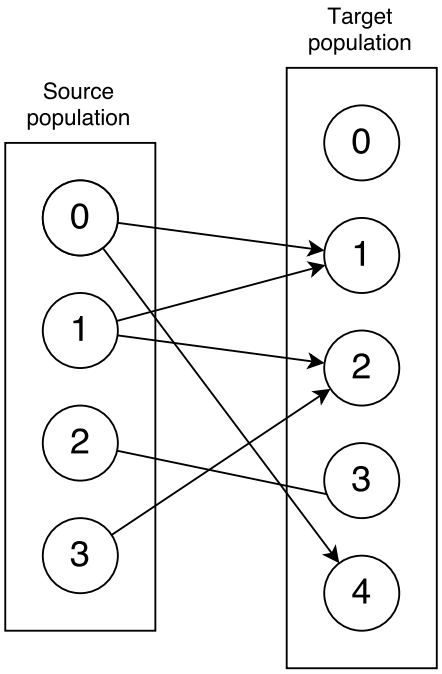
Full mask The full mask $\bar{\Omega}$ represents a connectivity pattern where every presynaptic neuron is connected to every postsynaptic neuron.

Empty mask The empty mask $\bar{\emptyset}$ represents no connectivity at all.

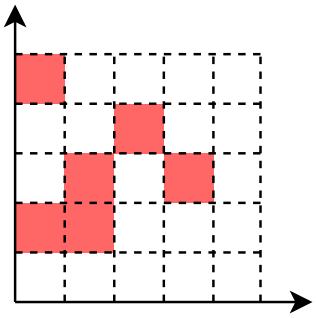
One-to-one The one-to-one mask $\bar{\delta}$ connects exactly one postsynaptic-neuron to every presynaptic-neuron in increasing order of IDs.

Random The random mask $\bar{\rho}(p)$ is a parametrized mask that connects the two neuron populations randomly. For every neuron combination a Bernoulli trial is performed to determine if a connection exists.

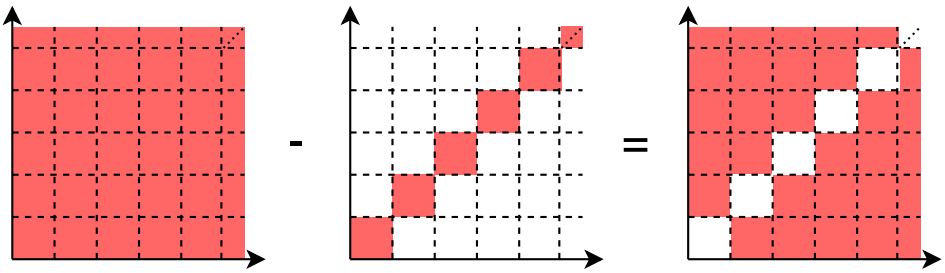
FanIn/FanOut The parametrized FanIn mask $\bar{\rho}_0(n)$ connects n presynaptic neurons to every postsynaptic neuron. The FanOut mask $\bar{\rho}_1(n)$ similarly connects n postsynaptic neurons to every presynaptic neuron.



(a) Connectivity pattern between two populations.



(b) A CSA mask for the connectivity in Figure 1.1a. The mask corresponds to the connection matrix.



(c) Applying the difference operator on the full mask and the one-to-one mask. This matches the CSA expression $\bar{\Omega} - \bar{\delta}$. The resulting mask is a full connectivity without self connections.

Figure 1.1: CSA masks as connectivity matrices.

All the elementary masks are infinite and work on any kind of population sizes. These elementary masks alone can cover most of the connectivity patterns commonly used in network models, and can also be found as connectivity modes in most simulators.

To create connectivity patterns that go beyond the simple elementary masks, the algebra offers set operations to combine the existing elementary masks to create new masks. The set operators are as follows:

- The Complement operator $\neg\bar{M}$ is a unary operator that returns a set which contains the connections that are not part of the original set $\bar{M}$. For example, the complement of a one-to-one mask $\neg\bar{\delta}$ would equal a full connectivity without self connections.

- The Intersect operator $\bar{M} \cap \bar{N}$ is a binary operator that combines two sets to only include connections that can be found in both sets. A one-to-one connection intersected with a random connection $\bar{\delta} \cap \bar{\rho}(p)$ would thus result in a pattern of randomly chosen self-connections.

- The Union operator $\bar{M} \cup \bar{N}$ is a binary operator for adding two sets together. If a connection is contained in either set, it will also be in the resulting set. Creating a Union between the one-to-one mask and the random mask $\bar{\delta} \cup \bar{\rho}(p)$ would then result in a random pattern where all self-connections are assured.

- The Difference operator $\bar{M} - \bar{N}$ is a binary operator for excluding certain connections. All connections in $\bar{M}$ are contained in the resulting set, except for those in $\bar{N}$. Subtracting the one-to-one mask from the random mask $\bar{\rho}(p) - \bar{\delta}$ would thus result in a random mask without self-connections.

Figure 1.1c shows an example of the difference operator. If the operands are infinite masks, the resulting masks will also be infinite. This allows masks combined through set operators to retain the population independence. Using these operators allows for creating complex connectivity patterns in an algebraic way.

Value Sets

The Connection Set Algebra can not only define which neurons to connect with a synapse, but also represent the parameters of the synapse. A connection's parameters can be described in generic value sets $\{V_0, V_1, \ldots, V_n\}$, where V is defined as a function $V : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{R}^N$. These values may refer to any parameters of the connection, usually the weight and delay of the synapse, but it may also refer to more complex parameters, such as different models of plasticity. The value sets are unnamed, and the user is responsible to relay the correct value sets to the correct parameter in the simulator. In its

basic form, value sets are unrelated to the mask in a connection set, but they may be combined using metrics. Using value sets gives the advantage of defining connectivity parameters alongside the actual connectivity, and keeping parameter definition independent of simulators.

Metrics

Although many connectivity patterns are completely independent from their populations, some patterns rely on information about the neurons they connect. For example, the spatial connectivity pattern requires the spatial distance between neurons for generating connections. CSA can supply this information using **metrics**. A metric is a function applied to the indices of neurons and it can be interpreted as a value set of the algebra. The distance metric for example can be defined as a function $d(i, j) = \|p_i - p_j\|$ where p is the positional vector for a neuron.

To create a mask based on a value set or metric, an **operator** needs to be applied to the metric. The operator takes a value set and parameters as input and maps them to a boolean value, where a true boolean indicates a connection between two neurons. The disk operator $\bar{D}(r)$ for example creates a mask that connects every neuron in the radius r. The operator has to be combined with the distance metric, which supplies the distances between all neurons. In the algebra, applying an operator to a metric is done by multiplying both elements from the right: $d(i, j)\bar{D}(r)$. Since both metrics and operators are arbitrary functions, the algebra can be easily extended to support any form of connectivity.

Implementation

The mathematical language that describes the Connection Set Algebra is separated from its implementations. The Python and C++ implementations of CSA serve as the connection generation libraries, which support the Connection Set Algebra, and are used to generate the actual connectivity. The Python implementation of CSA supports most of the features defined by the algebra in the Python package `csa`. Connection sets, masks and value sets are implemented in a class structure that provides elementary masks, intervals and metrics. The elementary masks are, for example, provided as `csa.full`, `csa.oneToOne` and `csa.random`. Using Python's basic operators, masks can be combined to create new mask objects. A union between two CSA objects can be written in Python as `m1 + m2`, a difference as `m1 - m2`, and an intersection as `m1 * m2`. The operators are built to rely on context: The classes perform different functions depending on the operator and the class they are combined with. For example, `csa.random` acts as a random

mask when combined with other masks, but when used with value sets it creates a mask by sampling from the value set.

All CSA objects can be exported to and imported from XML. This allows CSA structures to be passed between implementations of CSA or stored separately from the generating code.

Since Python is an interpreted language it lacks the performance of compiled languages. Generating connectivity with the Python implementation can be a slow process, and therefore is only recommended for small networks. The algorithms used for generating connectivity generally run in linear time, with the exception of spatial connectivity, which requires an algorithm with quadratic complexity (see Section 4.2.1). While the algorithms scale linearly, the set operations (union, difference, etc.) do add a lot of computational load to the generation process (see Section 3.2.4). The Python implementation does support multithreading, which can be used to increase performance on multiprocessing and distributed systems.

Currently in development for the connection set algebra is also a C++ implementation called `libcsa`. The C++ implementation shows a significant performance increase opposed to the Python implementation. The performance of both libraries is compared in Section 4.2. So far, the C++ implementation only has support for some elementary masks, like full connectivity or random connectivity. Value sets and set operators are not supported in `libcsa` as of yet. The C++ implementation can also be controlled using a Python interface written in Cython [2]. The interface tries to support the same structure as the Python implementation of CSA, while generating connections significantly faster. This makes including support for the C++ version an easy task in the future, as the interface for accessing CSA functions stay the same over all implementations.

1.3.2 Connection Generation Interface

To achieve the modularity needed for the project to work with any simulator, an interface between the simulator and the connection generation library is needed. Using an interface simplifies the development process of adding support for a simulator to the translator. The interface makes the connection generation process simulator independent. The Connection Generation Interface also allows switching out the Python implementation of CSA with the faster C++ implementation in a later stage of development. An interface would thus make development on both ends much easier.

To allow any simulator to benefit from connection generation libraries, the Connection Generation Interface (CGI) has been developed by Djurfeldt, Davison, and Eppler [8]. The CGI allows the simulator to query connections from the linked connection generator. Figure 1.2 shows how the CGI links a connection generating library to the simulator. For

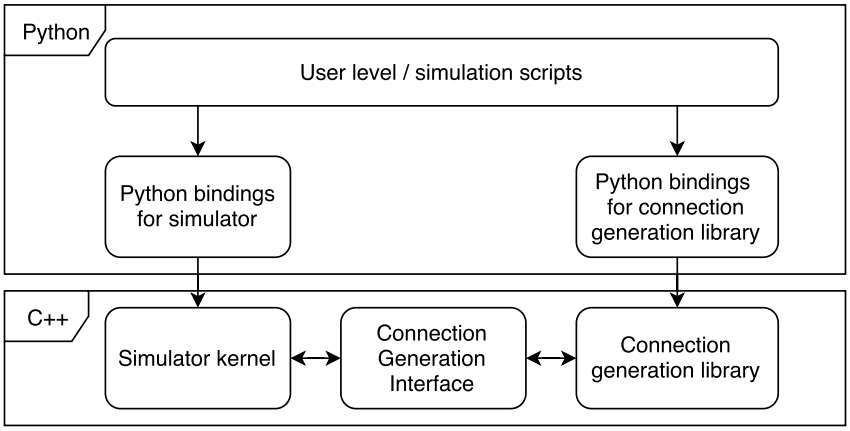


Figure 1.2: The CGI linking the simulator and the CG library. Figure taken from Djurfeldt et al. [8].

this both the simulator and the connection generator need to implement the interface. The interface describes the `ConnectionGeneration` class, which has to be implemented by the connection generator.

`int size()` returns the number of connections for this generator.

`int arity()` returns the number of parameters per connection, e.g. synaptic weight and delay.

`void setMask(Mask& mask)` sets the mask which determines which nodes are available for the connection generator.

`void start()` has to be called before starting an iteration.

`bool next(int& source, int& target, double* value)` passes along the next connection in the iteration, or returns false if there are no more connections.

With both parts connected through the CGI, the user can then use the simulator with the connection generation library. First, the user creates a new connection generator (e.g. from the Python CSA implementation) and connects it to the CG-Interface. Then the CG-Interface is called by the Simulator kernel to start the connection generation. The CG-Interface converts the global IDs given by the simulation kernel into interval masks and passes the masks to the connection generator. The interval mask is then iterated through by the simulator and the neurons are connected based on the connection generator's algorithm.

The interface has been implemented for the NEST simulator and the NEST backend in PyNN. It supports the two connection generation libraries for CSA, `csa` for Python and `libcsa`. The implementations can be found in the `libneurosim` package.

1.3.3 Neural Simulators

Simulators for spiking neural networks are a central part of computational neuroscience. A simulator allows the creation and simulation of complex neuronal models of different sizes. The simulators provide a language, usually a scripting language, for the user to access the network creation and simulation functions. With the simulator, the user can simulate single neurons, synapses, or networks of neurons.

The most common spiking neuron simulation tools are NEURON [17], NEST [20, 12], GENESIS [3], and BRIAN [15]. Tikidji-Hamburyan et al. [29] compares the performance and capabilities of these simulators. Of the four simulators, NEURON is the oldest and most used in the neuroscience community. NEST is the newest of the four simulators. While NEURON is used for more detailed models, NEST is best used for large scale models on high performance computers. BRIAN supports simulation of both detailed and large scale networks, but lacks high performance computing tools. All simulators have Python interfaces or can be controlled using a simulator language (e.g. PyNEST [9] for NEST).

Due to the many different simulators, PyNN [6] tries to aggregate existing simulators such as NEST and NEURON by providing a high-level Python interface. The interface can be used to control the different supported simulators, and experiments can be translated from one simulator to another. To achieve this, PyNN attempts to match the different neuron and synapse models of the simulators to each other.

An example for a specialized simulator is Topographica [1]. Topographica focuses on simulating large scale networks consisting of topographic maps. Instead of having single neurons form a network, Topographica simulates two-dimensional sheets of neurons as single entities. This results in better performance for large scale networks at the cost of accuracy. The main area of application for Topographica is the visual cortex.

1.3.4 NeuroScheme

In neuroscience, visualizations of network models can become quite convoluted. The visualization needs to adapt to the different sizes and levels of complexity of the network models. For this, Pastor et al. [25] have developed a visual exploratory framework called *NeuroScheme* for visualizing complex network models. This tool can be used as a base for visual connectivity generation.

NeuroScheme was developed as a visualization of morphological data in the neocortex area of the brain. The representation of the neocortex is multiscalar. In a multiscalar environment, elements can be observed at different abstraction levels. The levels of abstraction in NeuroScheme are (from lowest to highest): the Branch entity, where the

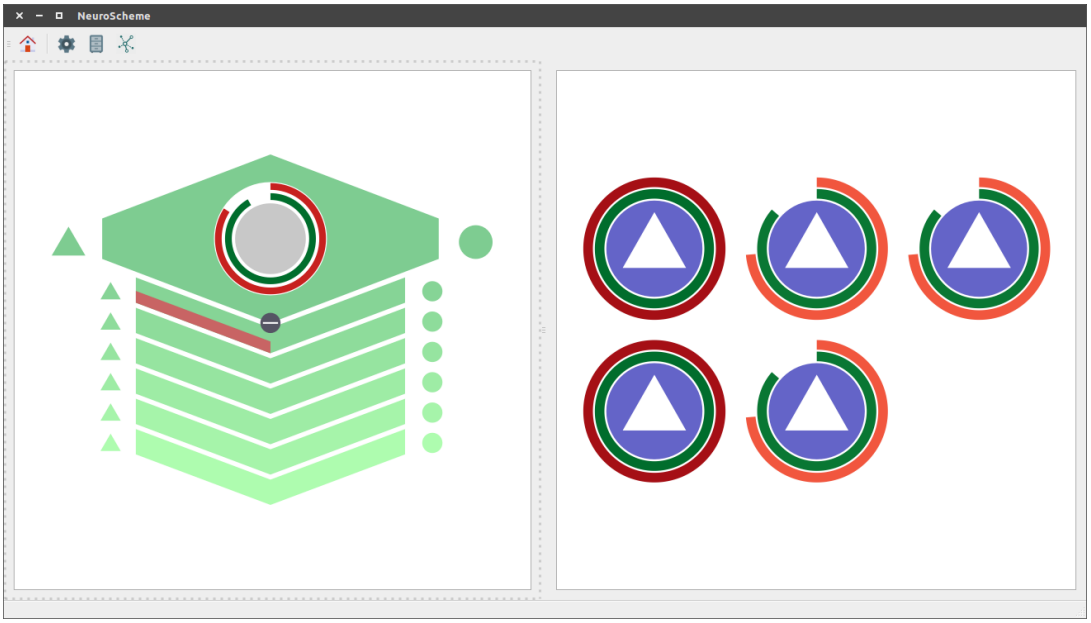


Figure 1.3: NeuroScheme's Graphical User Interface, showing a microcolumn on the left and its containing neurons on the right. The neuron is denoted as an excitatory neuron by the triangle in the center. The rings stand for the size of the soma and the length of the dendrites. Inside the microcolumn, the average neuron is displayed in the center. The bars on the bottom of the microcolumn show the distribution of excitatory (left) vs inhibitory (right) neurons on each of the six cortical layers.

dendritic tree of a neuron is displayed; the Neurite entity, which gives a statistical overview of the dendrites and axons of a neuron; the neuron entity itself, showing statistics of the cell and the dendrites; the Microcolumn entity, a collection of neurons along a cross-section of the cortex; and the Column entity, which is a collection of microcolumns. Each level encompasses entities from the lower levels, meaning that the user can "zoom into" a single element to see additional detail or the entities it is composed of. For example, the user can select a microcolumn entity and see the individual neurons that make up this microcolumn, as seen in Figure 1.3. The visualization of the entities all show different statistics that are either parameters of that object, or an average of the parameters of the lower level entities. The parameters are incorporated in the representation using icons, shapes and diagrams in the visual elements. The visual representations currently in NeuroScheme are developed specifically to fit the neocortex; for example a column shows the distribution of neurons on the six layers of the neocortex. To use NeuroScheme as the visual tool for generating connectivity, new visual elements have to be created for representing connectivity.

NeuroScheme is written in C++ and the Qt5 application framework. Qt5 enables cross-platform support and many graphical features needed for a visual tool. The project is structured with a modular design in mind. All of NeuroScheme's data structures

are packed in a *domain*, allowing for different domains to be developed independently. The domain can be defined in an XML file that contains entity definitions and their parameters. This makes extending NeuroScheme to a new domain easier.

NeuroScheme was chosen as the visual environment for this project as it already supports many of the elements needed and implementation of new visual representations is simple. The multiscalar aspect of NeuroScheme will also be a useful concept in the project's future, when more complex representations are needed.

1.3.5 NeuroML

To transfer the visual ontology to the translator, an XML file is used. Instead of developing a file format specifically for this purpose, using an already existing file format ensures that already existing standards are adhered and development cost is kept low. Supporting NeuroML also opens the possibility of connecting many existing tools to the translator.

To be suitable for this project, the file format needs to fulfill some requirements. First, it should be easy to parse the file in various languages. Since the translator is written in Python, and the visual tool in C++, language-specific file formats such as Python's `pickle` library are ruled out. Second, the file format needs to be human readable. Human readable file formats make error checking a lot easier, and the file can be inspected or edited by the user. This requirement excludes bytestream based file formats, which would have the advantage of smaller file sizes and faster parsing. Third, the scope of the file format should cover all the elements of a network model, including projections, populations, and inputs. The first two requirements can be fulfilled by an XML or JSON structure, as both are human readable and commonly used in many programming languages. One file format that fulfills all three requirements is NeuroML.

NeuroML is developed by Gleeson et al. [14]. The NeuroML standard is a XML based formalism that can describe networks at different scales. Furthermore, NeuroML is simulator independent and is supported by various neuroscience tools. The standard consists of three levels, which are built hierarchically; Elements that are defined on a lower level can be used on higher levels.

Level 1 – MorphML This level describes the neuronal morphology using 3D segments. It's scope is a single neuron and it's dendritic structure.

Level 2 – ChannelML On this level, the cell membranes and synapses are defined. Additionally to the elements in Level 1, it can define a single connection between two neurons in detail.

Level 3 – NetworkML The highest level formulates the synaptic connectivity between neurons using synapse and neuron definitions of the two lower levels. It also defines the external input to the network.

The three levels of NeuroML provide standards from a single dendritic branch to a whole network with populations and inputs. Many of the elements that are required for this project can be found in NeuroML's third level, NetworkML. NetworkML gives a formalism for the three main elements: populations, projections, and inputs. Populations describe a group of neurons in 3D space that share a cell type. The cell type can be defined more precisely using ChannelML if needed. The neuron group can either be a list of neuron instances, or an area where a given number of neurons are distributed in. The following XML structure would define a population with three neuron instances:

```
<population name="PopulationInstanced" cell_type="CellA">
    <instances size="3">
        <instance id="0"><location x="0" y="0" z="0"/></instance>
        <instance id="1"><location x="50" y="0" z="0"/></instance>
        <instance id="2"><location x="100" y="0" z="0"/></instance>
    </instances>
</population>
```

This XML structure would define a population template with 50 neurons that are randomly arranged inside a three-dimensional box:

```
<population name="PopulationTemplate" cell_type="CellA">
    <pop_location>
        <random_arrangement population_size="50">
            <rectangular_location>
                <corner x="0" y="0" z="0"/>
                <size width="0.5" height="2.5" depth="2"/>
            </rectangular_location>
        </random_arrangement>
    </pop_location>
</population>
```

Projections between populations can be defined in a similar way. In the XML attributes the populations to connect are reference by the populations string identifier. The connectivity of the projection can also be instance based or template based. The following XML structure shows a template projection with a random connectivity pattern:

```
<projection name="PopA-PopB" source="PopA" target="PopB">
    <connectivity_pattern>
        <fixed_probability probability="0.1"/>
    </connectivity_pattern>
</projection>
```

The patterns that are available for template projections are defined in the NetworkML standard:

`all_to_all` Connects every presynaptic neuron to every postsynaptic neuron (full connectivity).

`fixed_probability` Connects neurons randomly, with every possible connection sampled using a Bernoulli trial. The parameter `probability` gives the threshold for the Bernoulli trial.

`per_cell_connection` Every presynaptic neuron is connected to a specified amount of postsynaptic neurons. The postsynaptic neurons are chosen randomly, and the number of postsynaptic neurons for every presynaptic neuron is specified by the parameter `num_per_source`. The parameter `direction` can switch the direction from Pre-to-Post to Post-to-Pre, where every postsynaptic neuron is connected to a set amount of presynaptic neurons.

The available connectivity patterns are similar to the elementary masks provided by CSA (See Section 1.3.1) and cover basic use-cases for network models. When using template projections, NeuroML does not define the algorithm behind the connectivity patterns, nor does it provide an implementation. Tools that use NeuroML have to provide their own implementation, which may result in different connectivity based on the tool.

The NeuroML standard is defined in detail using XML Schema Definition (XSD) files. While XSD itself is written in XML, it is used to define the structure of an XML file. An XSD file describes which XML tags and elements are available, what attributes they support, and what and how many subelements an element can have. This allows XML files with a complex structure to be validated using the Scheme file, allowing for easier error detection in a file.

An alternative to NeuroML is NineML, a neuronal modeling language similar to NeuroML developed by Raikov et al. [27]. NineML also uses an XML based standard defined in XSD and focuses on definitions on the network level, making it simpler to use than NeuroML. NineML files can also be read using a provided Python parser. It also defines projection and population elements, although no input elements. Projections and populations can also be defined in a template based mode, and NineML provides similar connectivity pattern definitions to NeuroML. Although NineML shows a lot of similarity to NeuroML, many lower level elements from NeuroML are not included in NineML, including the biological cell structure of the neurons and synapses. While NineML would also be a suitable candidate for the file format, NeuroML defines more elements that might be useful in the future of the project. NineML also has a lower adoption rate in

the neuroscience community, making the project less likely to be compatible with other programs. For these reasons NeuroML was chosen as the file format for exchanging model data between the visual tool and the translator.

1.3.6 Visualizations and User Interfaces for Neural Networks

Apart from NeuroScheme (see Section 1.3.4), there have been previous attempts at model visualization. *VisNEST* by Nowke et al. [24] provides analysis tools for exploring simulation data in a 3D environment. VisNEST also includes visualizations for connectivity, but does not provide tools for creating models. The focus in VisNEST is on the spatial layout of network models in 3D space and the spiking activity in them.

The NEURON simulator has a few different visualization tools. *ModelView* allows the user to explore a network through a tree-like data structure, and can display the dendritic trees of a neuron in 3D space. It does not have tools for exploring the simulated data. The *Network Builder* tool for the NEURON simulator allows interactive network creation. Through the Network Builder, a user can control NEURON's network creation routines using a GUI. Individual neurons can be placed into a scene and connected with lines. The visualization is simple, with neurons represented by a single letter, and connections being lines between them. Populations and projections are not supported.

Topographica's GUI focuses on its two-dimensional maps. The GUI can create and visualize networks during simulation, but is very limited due to the simulator's specific area of application. Layers are presented as gray-scale images to show the activity of the neurons or weight of the connections.

Visualizations like the *BrainNet Viewer* [30] focus on a topographic representation of brain regions. In these representations, the brain can be explored in different 3D environments. The BrainNet Viewer can show the surfaces and volumes of the brain or display regions as interconnected nodes. Another 3D visualization tool is *neuroConstruct* [13]. Instead of just visualizing models, neuroConstruct can be used to create models in a 3D environment. Complex spatial connectivity patterns can be created and models can be exported into simulation scripts for various simulators. While a three dimensional visualization may be helpful for analyzing populations in relation to their biological position in the brain, the 3D visualizations can become confusing when viewing complicated networks. 2D visualizations usually do not try to visualize spatial positions. For example, the tools proposed by Nordlie and Plesser [22] specifically target visualizations of connectivity through the use of connectivity matrices. The matrices are similar to those described in Section 1.3.1, and can also describe different parameters with color scales.

Chapter 2

Design

To solve the problem of translating a visual language, a tool chain of different software has to be built. This package will be called **ViCoGen**, short for **Vi**sual **Co**nnectivity **Gen**eration. In this chapter, the concepts and design for ViCoGen will be introduced. This includes conducting usability interviews of neuroscience experts, constructing user stories, and designing the workflow of the ViCoGen package.

2.1 Usability Interviews

To develop a visual tool for neuroscientists that is intuitive to use and displays complex information, a visual representation of connectivity has to be designed. To achieve this, a series of usability interviews with neuroscientists and possible users have been conducted. The participants were four neuroscientists from the INM-6 at the Forschungszentrum Jülich, and one computational neuroscientist at the Institut für Neuroinformatik at the Ruhr-Universität Bochum. The chosen participants are active in the field of synaptic plasticity research, recurrent neural networks, network synchronization, and large scale visual cortex simulations.

The five interviews were conducted over the span of four weeks and had a length of 30-40 minutes. The interviews had the following structure: First, the participant explained their field of work and their current relevant projects they were working on. The interviewer then explained the current project and how it could relate to their field. Next, a list of general questions about connectivity and connectivity parameters was asked, which generally followed these questions:

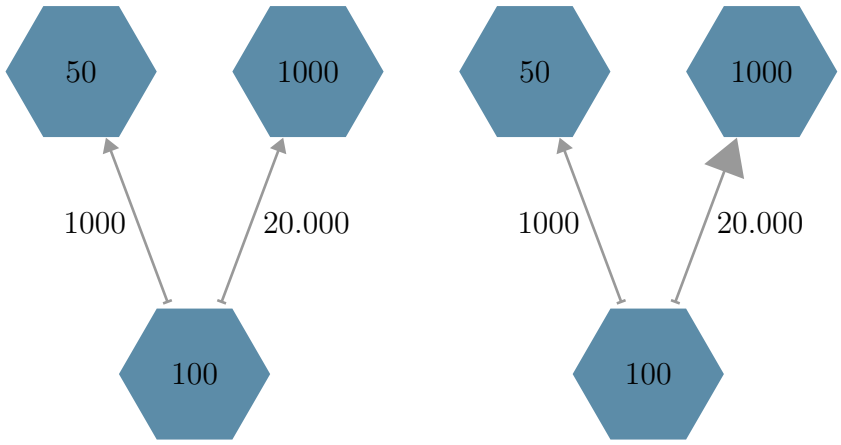
- Connectivity
  - What different types of connectivity are you using?
  - How sparse are the connections? / How many are there usually?

- Parameters
  - What parameters are important for your connections? (e.g. synaptic weight, delay, etc.)
  - Which parameters are most important to see at a first glance
  - Which parameters are changed most often? / What do you play around with most often?
  - How would you represent these parameters visually?
  - Do you use plasticity in your models?
- Spatial Connectivity
  - Do you use spatial connectivity?
  - Is the spatial connectivity two-dimensional or three-dimensional?

The participants were asked to support some of their answers with drawings and Mock-Ups. The rest of the time was used as a free-form interview, where participants could present their ideas about the project. The interviewer also presented Mock-Ups and asked for the participant's thoughts about the design.

The interviews were evaluated to find use cases and a common visual representation of connectivity. Most of the interviewed scientists use either a random connectivity, or a fixed per-cell connectivity (also called *fan in/out* or *fixed in/outdegree*) for their network models. Sometimes a full connectivity is used, one-to-one connectivity is almost never used. How many connections and neurons are used was widely different from project to project. Some used fewer than 100 neurons and connections, while others had connection counts in the billions. The connectivity percentage of layers was generally below 30%. Regarding the parameters, most people interviewed agreed that connectivity percentage and synaptic weight are the most important parameters to see in a network. Connectivity type is not as important to see, as most models use only one type of connectivity for all connections. Since they rarely differ in a single model, it is not important to visualize the connectivity for every connection.

When asked how to visualize a connection between two populations, all of the participants used a form of arrow to connect two populations. One participant preferred to draw individual neurons of a population that were then connected individually with lines, as the spatial representation of the neurons was more important to him. The direction of the arrow always pointed from the presynaptic neurons to the postsynaptic neurons. While an arrow tip was the usual indicator for excitatory connections, inhibitory connections were often indicated by a circle at the end of the line. Many participants agreed



(a) Connectivity Percentage (b) Indegree

Figure 2.1: The problem of connectivity percentage visualization illustrated in two examples. A small neuron population is connected to a small and a large neuron population with a random connectivity of 20%. On the left the thickness of the arrow represents the percentage of possible connections. Even though the connection between the small and the large population has more absolute connections, the arrow is the same size as the connection between the two small populations. On the right, the thickness is determined by the indegree of the connection. Here the arrow is intuitively larger for the small-to-large connection.

that the thickness of the line and the size of the arrow tip of an arrow representing a connection should be influenced by the synaptic weight and the connectivity of the projection. One approach to defining connectivity would be to calculate the percentage of actual connections to maximum possible connections. One participant concluded that the connectivity percentage would not be an intuitive representation of the number of connections, as it would not take neuron population size into consideration. Figure 2.1 illustrates this problem. The solution the participant suggested was calculating the *indegree* of a connection. The indegree is the product of the sparsity of the connection and the number of neurons in the target population. The connectivity visualization using indegree is further explained in Section 2.2.

When asked about spatial connectivity, the answers varied. Some of the interviewed scientists do not use spatial connectivity at all, while others used two- and three-dimensional spatial connectivity. Plasticity and synaptic mechanisms were also topics with multiple answers. Both synapse models and plasticity have highly varying parameter sets depending on the used model. Finding and implementing a visualization that encompasses all models and parameter spaces would be futile work, as many mechanisms do not share common parameters.

One feature that was requested by multiple participants in the free-form part of the interview was a display of connectivity in the form of a connectivity matrix. Many researchers use connectivity matrices as a way to display all connectivity data in a complex

model with a high number of populations. This feature would be an alternative to the more intuitive display, mainly to be used when the model becomes too complicated to be viewed with arrow connections.

Based on the analysis of the interviews, a design for the visual elements has been drafted in Section 2.2.

2.2 GUI and Visual Language Design

Based on the usability interviews, a set of requirements for the visual language was compiled:

V1: Projection source and target. The source and target of a projection should always be clear. It should also be easy to see which projections are connected to a population in a complex model.

V2: Inhibitory and excitatory. It should always be clear to see if a projection has a positive synaptic weight (excitatory connections), or a negative synaptic weight (inhibitory connections).

V3: Connectivity. The connectivity of a projection should always be visible. This connectivity should represent the indegree of a projection towards a population.

V4: Synaptic weight. The strength of the synapse should be visible for all projections. If a projection's synaptic strength is drawn from a distribution for all its connections, the mean of that distribution should be visible.

V5: Delay. The delay of a projection does not usually differ in a model, so the delay does not need to be visible directly.

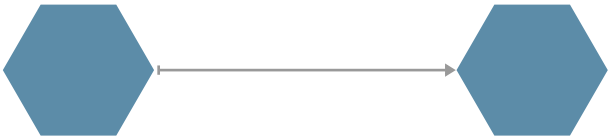
V6: Connection Pattern. The connection pattern of a projection is usually the same for all projections in a model and does not need to be always visible.

The resulting visual language derived from the requirements was developed in close collaboration with the GMRV in Madrid, as their team would be implementing the visualization.

Displaying a projection with a clear source and target population (V1) can be done with an arrow pointing from the source to the target (Fig. 2.2a). The arrow points towards the target population of the projection, indicating the direction of spikes in the connections. The arrow representation was unanimously used during the usability interviews, making it the most intuitive representation. Nowke et al. [24] also use arrows

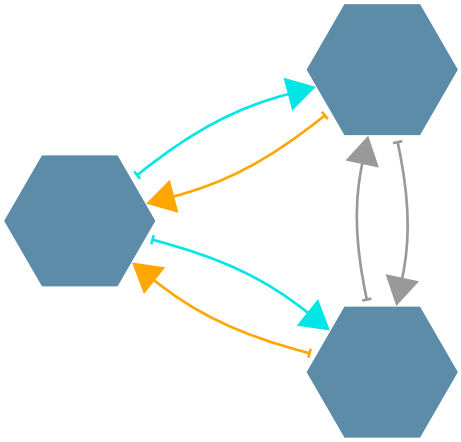
(a) Projection with excitatory connections, represented by a triangular arrow tip.

(b) Projections with inhibitory connections, represented by a circular arrow tip.



(c) Projection with low connectivity. The size of the arrow tip decreases with lower indegree.

(d) Projection with high synaptic strength. The thickness of the line increases with the synaptic strength.



(e) Fully connected network with 3 populations. The left population is highlighted, showing incoming projections in orange and outgoing projections in blue.

Figure 2.2: Examples of the visual language for connectivity. Neuron populations are displayed as blue hexagons, similar to NeuroSchemes columns.

in their 3D visualization of connectivity in VisNEST. Additionally, the start of the arrow is decorated with a small perpendicular bar, which makes it easier to differentiate the source population. To allow seeing at a quick glance which projections belong to a population, incoming projections should be highlighted in orange, and outgoing projections in blue (Fig. 2.2e).

Differentiating between inhibitory and excitatory connections (V2) can be done through the shape of the arrow tip. Normal arrow tips are used for excitatory connections, and circles for inhibitory connections. This is the representation used in Potjans and Diesmann [26, Fig. 1], and was also suggested during the interview process. Fig. 2.2a and Fig. 2.2b show the difference between excitatory and inhibitory projections.

The connectivity of a projection (V3), calculated as the indegree of the target population, is represented by the size of the arrow tip (Fig. 2.2c). As stated in Section 2.1, the indegree of a projection shows the amount of target neurons a projection connects to. This allows the user to see how sparse a connection is in relation to the number of target neurons. Since the indegree is dependent on the number of neurons, the size of the arrow tip should be scaled logarithmically to the indegree. Logarithmic scaling would allow comparisons between projections targeting small populations and projections targeting large populations. To avoid the size of the arrow tip to become too large, all arrow tip sizes should be relative to the largest indegree in the model.

The synaptic weight of a connection V4 is displayed using the thickness of the line (Fig. 2.2d). Connections with a high synaptic weight have a higher influence on the target neurons, and are thus represented by a thicker line. Since inhibitory connections have a negative weight, the absolute weight is used for the calculation. The weight of a connection is usually bound to a maximum value g_{max} by the synapse model. Values greater than the maximum are mathematically possible, but their strength would not be physiologically correct. Thus, the width of the line can be mapped linearly from the interval $[0, g_{max}]$. When the synaptic weight is given as a distribution, the average of the distribution should be used for calculation in the visual representation.

The color of the projection is intentionally not used for a specific parameter. Instead, it can be used to optionally display any generic parameter that may be a part of the synapse model or a measurement. Numerical parameters can be displayed as a color gradient, and multiple choice parameters as a random color according to a legend. Since visualizing the delay (V5) and the connectivity pattern (V6) is not as important, they should be handled like generic parameters.

Section 4.3.1 shows how the GUI design has been implemented so far in NeuroScheme.

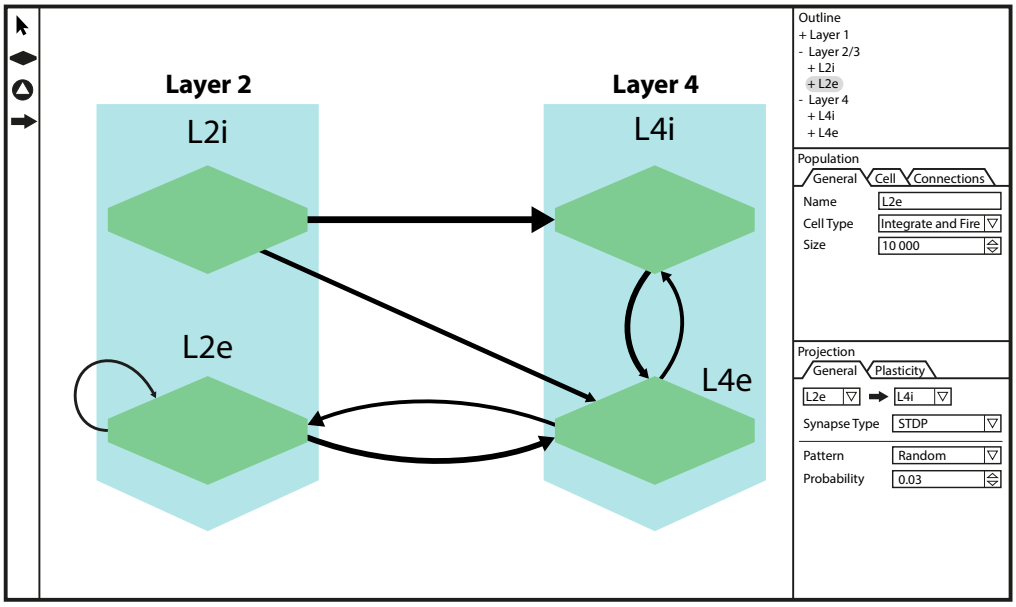


Figure 2.3: A Mock-Up design of the ViCoGen GUI. In the middle of the screen is the current scene, which contains the visual representations in a 2D space. The neuron populations, represented by hexagons, are grouped into layers and connected between each other with projections. On the left is a toolbar with access to tools for selecting and creating populations, and drawing new projections. On the top right is an outline which shows the populations and projections in the scene in a tree structure. Below that is the population properties panel, where population settings can be adjusted. Similarly, below the population properties is also a panel for setting projection properties.

2.3 User Workflow

To ensure an intuitive user experience, the GUI has to be designed with the target user base in mind. ViCoGen is made to be used by neuroscientists and students of neuroscience; no programming experience should be needed to use the visual tool. This section will introduce a number of user stories on how the visual tool can be used to create new models. The user stories are based on the Mock-Up design shown in Figure 2.3.

When the user starts the tool, he is presented with an empty scene. On the left is a selection bar for tools: *Select*, *Create Population*, *Create Neuron*, *Create Connection*. In the center is the current scene, which shows created elements in their visualizations. On the top right is an outline of all the created elements in the scene for quick selection and inspecting the hierarchy. The bottom two panels on the right show the properties of the currently selected population and connection. Created populations can be placed freely in the scene. Positions of the population elements do not infer the spatial positions of the neurons in the population.

The following user stories represent the basic usage of the tool and include creating and simulating models, modifying existing models, and searching elements in a model:

U1: Create neuron population. The user wants to create a new population from an empty scene. He either selects the population creation tool from the left toolbar and clicks into the empty scene, or right clicks the empty scene and selects the option for creating a new population from the context menu. The new population is selected after creation and the user can adjust parameters, e.g. number of neurons or the neuron model, in the panel on the right. The default parameters of the new population are copied from the last population that was created, making it easier to create multiple similar populations.

U2: Connect two populations. The user wants to create a new projection between two existing populations. He selects the connection tool in the left toolbar, and drags a new projection from the source population to the target population. Similar to the population creation, the new projection is selected and the parameters can be adjusted on the right. The parameters change the visualization of the projection in the scene, as described in Section 2.2.

U3: Edit/Delete a projection The user wants to change the parameters of a projection or delete the projection entirely. He switches to the selection tool and clicks on the projection. The projection is highlighted and its parameters appear in the panel on the right. He can now change the parameters of the projection, or delete it with a button in the parameter panel or by pressing the `Del` key.

U4: Connect input to a population. The user wants to connect an input generator to a neuron population. He selects the Input tool from the left toolbar and clicks into the empty scene to create a new input. In the properties panel on the right he can adjust the type and parameters of the input. He then connects the input to a population using the connection tool. The type of connection that can be selected may be limited by the source input's type.

U5: Search for a specific population. The user is working on a model with a large amount of populations and wants to adjust the parameters of a specific population. He uses the search function in the outline panel on the top right to type in the name of the population. The outline is filtered to show the populations that match the search terms. The user can select the population by clicking on it in the outline, or focus on the population in the scene by double clicking.

U6: Change synapse parameters of all outgoing projections of a populations. The user wants to adjust the parameters of all outgoing projections of a population. He selects the population, which highlights every incoming and outgoing projection.

He can now select multiple projections by control clicking in the scene. Alternatively, the selected population is also selected in the outline. When expanding the population in the outline, it shows links to all ingoing and outgoing projections. The user can select the category for outgoing projections to select all at once. The parameter panel on the right can now be used to adjust the properties of all selected projections.

U7: Change the connectivity method of all connections. The user wants to change the connectivity method of all existing connections. He uses the command `Ctrl+A` to select everything in the model. He selects the projection properties panel on the right and changes the connectivity method. Some parameters may be converted between the connectivity methods, but the user will have to edit each connection's parameters individually for other connectivity methods.

U8: Grouping populations. The user is working on a complex model and wants to group some populations into layers for better overview. He selects three layers using control click and groups them together using either the right click context menu or by using the `Ctrl+G` shortcut. The population group is indicated by a gray rectangle encompassing the populations. The group can be moved similarly to other elements, with populations in the group moving with it.

U9: Entering multiple parameters. The user wants to implement an existing model that is defined in a paper. The model has a high amount of projections, so connecting the populations by hand is time-consuming. Instead, the user switches into the connection matrix view, where parameters are ordered into a matrix. The columns of the matrix lists the outgoing projections of a population, while the rows shows all incoming projections. The user selects the connectivity percentage parameter from a list and can now enter the parameters similar to a spreadsheet. The individual cells of the matrix can also be highlighted in a color grade linked to the parameter value.

U10: Simulate the model. The user has finished designing his model and wants to simulate it using NEST. He selects the *Simulate* option from the menu bar, which opens the simulation dialog. In the dialog the user selects the NEST simulator as a target simulator, which also gives him a list of parameters specific to NEST. He selects the simulation time, the number of threads, the output target, etc. in the dialog and then clicks the *Simulate* button to start the simulation.

Section 4.3.1 shows how far the implementation of the visual tool by the GMRV is.

2.4 Technical Requirements

At the backend of ViCoGen stands the translator. Its purpose is to translate the models created in the visual representation into a simulation specific language. The translation system should support the following technical requirements:

T1: The system should be able to simulate a model provided by the GUI in a simulator.

T2: The simulation should be able to be performed in different simulators and adding support for a new simulator should require a low development cost.

T3: The system should be separated of the GUI.

T4: The overhead of the system should be low. Performance should be close to that of using the simulator directly.

T5: Calculation on distributed systems should be supported.

The requirement T1 is the main task of the translator, and should be considered the critical requirement. Requirements T2 and T3 are important for ensuring the modularity of the system. T2 would allow simulators to be used as modules for the backend. T3 in turn makes the frontend exchangeable. T4 and T5 exist to ensure the performance of the system. How the technical requirements are solved can be found in Chapter 3.

Chapter 3

Implementation

This chapter will explain how the requirements gathered in Chapter 2 are implemented, how the technology is applied, and how each part of the project can be used.

After a user has created a model in the visual tool, he can start simulating it. For this, the model is parsed through the backend of ViCoGen to a simulator. Figure 3.1 shows the flow of data from the visual tool to the simulator, and how the modules split into frontend and backend.

The backend of ViCoGen is designed to be separate from the GUI, as described in T3. This separation allowing the frontend and backend to be used independently as modules. The frontend can for example run on a desktop computer, while the more performance costly backend runs on a high performance computing system. This design principle of modular design also allows for single parts to be rewritten or switched out easily. New parts can be developed independently by external teams, and developers only need knowledge about the part they are working on, decreasing the time it takes to introduce a new developer to the project. Developers are also able to work on the project simultaneously on different modules without affecting others. Parallel development is especially important in this project, as the ViCoGen frontend is developed in parallel by the GMRV in Madrid. Disadvantages of a modular design are the added overhead to development and the increased complexity of data transfer. Since the project involves multiple individual teams and future development is likely to be carried out by different master's students, a modular design has been chosen for the project. Using the modular design principle requires careful construction of interfaces and data exchange points.

The main data exchange point between the frontend and the backend is done through NeuroML. The visual tool exports its visual data into a NeuroML file, and passes the file along to the translator. When the GUI is running on the same system as the backend, the GUI can start the translation process immediately. The NeuroML file can also be

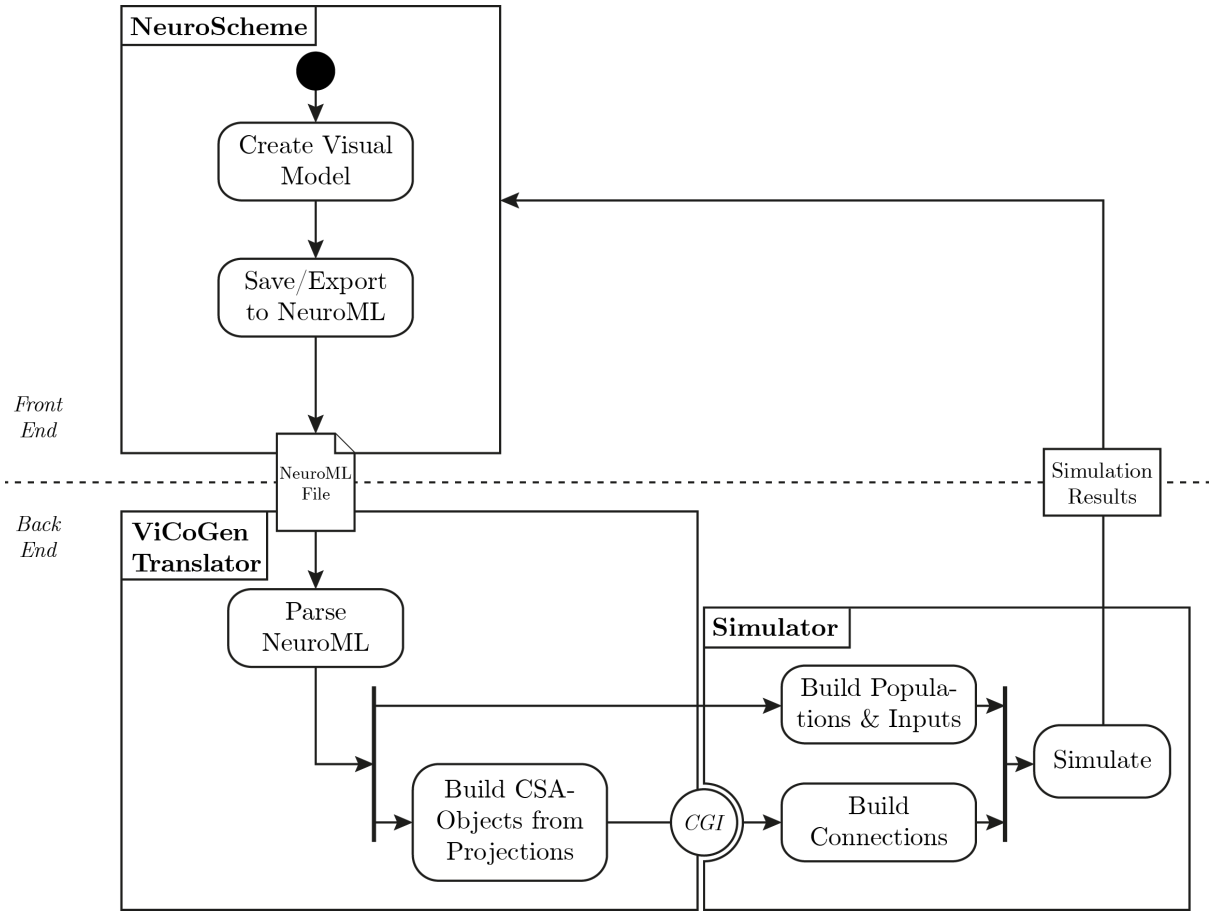


Figure 3.1: Flow of data from NeuroScheme to the simulator. The user creates a model in NeuroScheme and exports it as a NeuroML file. In the backend, the translator parses the NeuroML file, and converts the connectivity into the Connection Set Algebra. The populations and inputs are built directly in the simulator. With the CGI, connections are generated from the connection generation library and passed to the simulator. The simulator starts the simulation, and the results can be processed and passed back to NeuroScheme.

saved and then translated on a different system, making the frontend completely separate of the backend.

The main part of the ViCoGen backend is the translator. The translator parses NeuroML files that are generated by the frontend, as described in T1. To pass the model data to the simulator, the translator uses CSA and the Connection Generation Interface to translate the connectivity. With the CGI, the link between translator and simulator also becomes modular.

3.1 Extending NetworkML

For this project, NeuroML version 1.8.2 was used, with a version 2.0 currently in development. Version 2.0 does not change much on the NetworkML layer, but uses the LEMS XML language as a framework for defining elements [5]. As NeuroML 2.0 is still in development, ViCoGen uses NeuroML 1.8.2 as a standard. To support all visual requirements defined in Section 2.2, the NeuroML file format had to be extended, specifically NetworkML. These extensions are entirely optional and files of the original NeuroML standard can still be parsed. The XSD-Scheme file can be used to check a NeuroML file for validity using external programs. The NeuroML file is not checked for validity by the translator, as the NeuroML files are not supposed to be written by hand but returned from other parts of the project. Instead, the validity of the file should be checked by the parts that create the file. The translator may be able to parse malformed NeuroML files, but no guarantee is given. The following sections describe the changes and additions made to the NeuroML specification.

3.1.1 One-To-One Connectivity

Of the common elementary connectivity patterns, NeuroML is missing the One-To-One connectivity. Since a one-to-one connection does not need any parameters, adding it to the XSD Scheme is trivial:

```
<xs:element name="one_to_one">
    <xs:complexType/>
</xs:element>
```

While the One-to-One connectivity may not be common for biological networks, it is important for connecting inputs and recording devices.

3.1.2 Input Sites

In NeuroML, an input is connected to a population by defining an input site. The sites function the same way as projections, as they can connect an input to a population using either a connectivity pattern or connection instances. Since using an input site is the same as connecting an input through a projection, this creates an unnecessary redundancy in the file structure. New connectivity patterns would have to be created for projections as well as for inputs. To avoid the redundancy and additional workload, input sites are not supported by the parser. Instead, inputs are connected the same way two populations would be connected: With a projection where the source is interpreted as a population with size 1. This makes the file format easier to maintain and expand, and also simplifies the parser.

3.1.3 Spatial Connectivity

Of the interviewed scientists, almost half said they use spatial connectivity in their models. The most common variant of spatial connectivity is Gaussian spatial connectivity (Fig. 3.2). To support this, two new XML structures for Gaussian spatial connectivity in 2D and 3D have been added to the NeuroML file format.

```
<xs:element name="gaussian_connectivity_2d">
    <xs:complexType>
        <xs:attribute name="sigma" type="xs:decimal"/>
        <xs:attribute name="cutoff" type="meta:NonNegativeDouble"/>
    </xs:complexType>
</xs:element>
<xs:element name="gaussian_connectivity_3d">
    <xs:complexType>
        <xs:attribute name="sigma" type="xs:decimal"/>
        <xs:attribute name="cutoff" type="meta:NonNegativeDouble"/>
    </xs:complexType>
</xs:element>
```

Both XML elements have the parameters `sigma` and `cutoff`. The `sigma` parameter controls the size of the Gaussian that is used for calculating the probability of a connection. The `cutoff` parameter can be used to set the maximum distance a point can have to the center of the Gaussian. With this addition to the XML structure, a Gaussian spatial connectivity can now be given as follows:

```
<connectivity_pattern>
    <gaussian_connectivity_2d sigma="1.5" cutoff="3"/>
</connectivity_pattern>
```

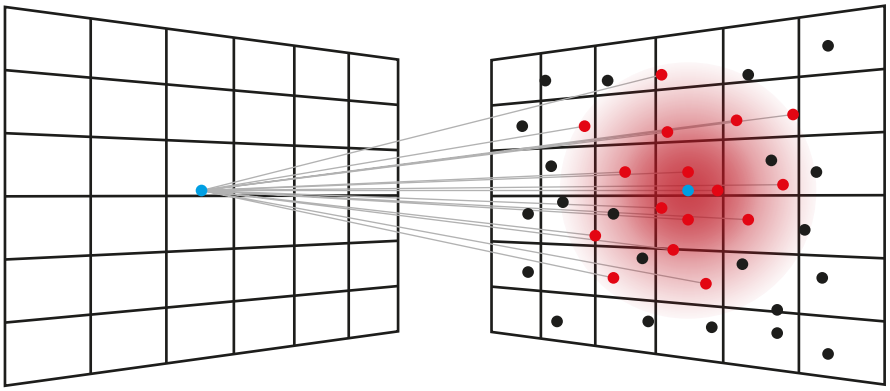


Figure 3.2: Gaussian spatial connectivity on two 2D-Layers. The neuron to connect from the source layer gets projected onto the target layer using an arbitrary mapping function (blue dots). The distance metric from CSA (See section 1.3.1) gets applied to a Gaussian sampler (red circle), resulting in Gaussian local connectivity (red dots).

The positions of the neurons can already be defined in NeuroML by either giving a position element to all neuron instances, or by adding a `<pop_location>` element to a template based population. The neuron locations in NeuroML are always three dimensional. When using 2D spatial connectivity, the value for the `z` dimension is ignored.

Adding these two structures provides simple spatial connectivity for 2D and 3D populations, but more advanced forms of spatial connectivity are still missing. Additional features may be scaling and translating of neuron positions, or even generic projection functions. Generic projection functions are supported in CSA, but have no equivalent in NeuroML.

3.1.4 Distributed Synaptic Parameters

When describing connectivity parameters, e.g. synaptic weight or delay, NeuroML only provides static values. Only instanced connections can have individual parameter values. For connectivity patterns though, NeuroML provides no way of giving each connection in the connectivity pattern a different value. All connections in the connection pattern share the same static value for their parameters. A solution to this is using distributions for parameters. To accomplish this, a new `DistributedProperty` complex type has been added to the XSD Scheme. A distributed property can hold one of the defined distributions: `GaussianDistribution` and `UniformDistribution`.

```
<xs:complexType name="DistributedProperty">
    <xs:choice>
        <xs:element name="GaussianDistribution" type="GaussianDistribution"/>
        <xs:element name="UniformDistribution" type="UniformDistribution"/>
    </xs:choice>
```

```
</xs:complexType>

<xs:complexType name="GaussianDistribution">
    <xs:attribute name="center" type="xs:decimal" use="required"/>
    <xs:attribute name="deviation" type="xs:decimal" use="required"/>
</xs:complexType>

<xs:complexType name="UniformDistribution">
    <xs:attribute name="lower" type="xs:decimal" use="required"/>
    <xs:attribute name="upper" type="xs:decimal" use="required"/>
</xs:complexType>
```

A distributed property can be defined inside a `synapse_props` block of a projection. So far, only *weight* and *delay* are supported, as the CGI implementation in the `libneurosim` package only supports weight and delay parameter for now.

```
<xs:all minOccurs="0">
    <xs:element name="weight" type="DistributedProperty"/>
    <xs:element name="delay" type="DistributedProperty"/>
</xs:all>
```

The Gaussian distribution defines a center and a deviation, which correspond to μ and σ of a Gaussian. The uniform distribution has an upper and lower bound and gives an even probability for every value between these bounds. Adding a new distribution can be done in XSD by adding it to the `DistributedProperty` complex type. New types of distributed parameters can be added in the `SynapseInternalProperties` complex type as a `DistributedProperty`. An example for distributed parameters as child elements in a NeuroML file can be seen here:

```
<synapse_props synapse_type="StaticSynapse" threshold="-20">
    <weight>
        <GaussianDistribution center="87.8" deviation="8.8"/>
    </weight>
    <internal_delay>
        <UniformDistribution lower="0.5" upper="1.5"/>
    </internal_delay>
</synapse_props>
```

The distributed properties are entirely optional and synapse properties can still be defined as an attribute of `<synapse_props>` instead of a child element. Alternatively, a static value may also be given as a child element of `synapse_props` for weight and delay, making it a possible replacement for the old way of writing parameters. It would have been possible to encode the distributed parameters in the string attributes of the `synapse_props` element, which would have required no changes to the XSD, but this would also make it impossible to check the validity of the elements using XSD. Additionally, encoding distributions as

a string would introduce a structural inconsistency to the NeuroML format which would make it difficult for parsers to correctly parse the NeuroML files.

These NeuroML extensions were necessary as NeuroML's connectivity definitions for large networks is limited. Dynamically creating connectivity patterns similar to CSA is not possible in NeuroML, and distributed properties are not supported. The only way to recreate spatial connectivity or distributed synapse parameters is by computing the patterns or distributions before writing the NeuroML file, and using NeuroML's connection instances instead. This would, especially for large networks, increase the file size of the NeuroML files dramatically. NineML does inherit these problems, although it does support distributed parameters.

3.2 Translation

To transfer the model information into a neuronal simulator, a translator is needed. This translator connects the visual tool with the simulator by using the Connection Generation Interface, CSA and the NeuroML file format. As seen in Figure 3.1, the translator parses NeuroML files, translates the connectivity of the model into CSA structures, and then passes the connectivity through the CGI to the simulator. The translator reads the information from the model file or gets called by NeuroScheme directly. Writing the translator was the main programming challenge of the project. As a target language, Python version 2.7 was chosen, as the NEST and CSA interfaces are also written in Python. The modules have also been made Python 3.6 compatible.

3.2.1 Python Module Structure

All of the python files are contained in a package named `vicogen`. The NeuroML files are handled in the `neuroml` subpackage. The modules in the `neuroml` package are for holding the data structures of the translation and parsing NeuroML files. The module `parsing.py` contains all the functions for parsing NeuroML files into the data structures. The `vicogen` package itself contains a few different modules:

- `_vicogen.py` contains functions for accessing the file parsing from the `neuroml` package, starting simulations with NEST and printing connectivity matrices and lists to the console or to file. All of the functions in this module are made available on a package level.

- `nest.py` is the module responsible for connecting the translator to the simulator NEST. It provides functions for translating populations and calling the CGI-Functions for NEST, and also starting simulations.

- `__main__.py` is a required file for using the package by itself on a command line. It allows the package to be called like a module using `python -m vicogen PARAMETERS`.
- `commands.py` provides the command arguments that can be used when calling the module from a command line.
- `__init__.py` is called when `vicogen` is imported as a python package. It gathers all the functions from the other modules and provides them on a package level.

The `vicogen` package can be used either as a normal python package using `import vicogen` or executed as a python file with command lines for quick access (See supplementary material in Section 6.3).

Coupling

The modular nature of the translator allows for extensibility and reuse of the different parts in the translator. The modules in the `vicogen` package are written to ensure a low coupling between each other. For example, new simulators can be added easily as all references to NEST are contained in the `nest.py` module.

The modules in the `neuroml` package do not have a very low coupling, as the data structures have to be interconnected to function properly. Instead, the modules are parted into semantic groups to ensure readability. Still, the `neuroml` package can be used independently as it does not depend on any part of the `vicogen` package. The `neuroml` package can therefore be used in other projects for parsing NeuroML files.

3.2.2 Class Structure

To represent NeuroML in Python in a way that is extensible and maintainable, the class structure makes use of inheritance and "duck typing". For example, all connectivity pattern classes are subclasses of the `ConnectivityPattern` class, which has an empty function `mask()` that all subclasses have to implement (See Section 3.2.4). This makes it easier for developers to see which functions they need to implement to add a new pattern. Fig. 3.3 shows a class diagram of all the classes represented in the `neuroml` package. All connectivity patterns are subclasses of `ConnectivityPattern` and `Projection`. Adding a new connectivity pattern can be done by creating a new subclass of `ConnectivityPattern` and assigning a CSA mask to `self._mask` in the `__init__` method. Similarly, new subclasses can be created to add new distributions, neuron position functions, and inputs.

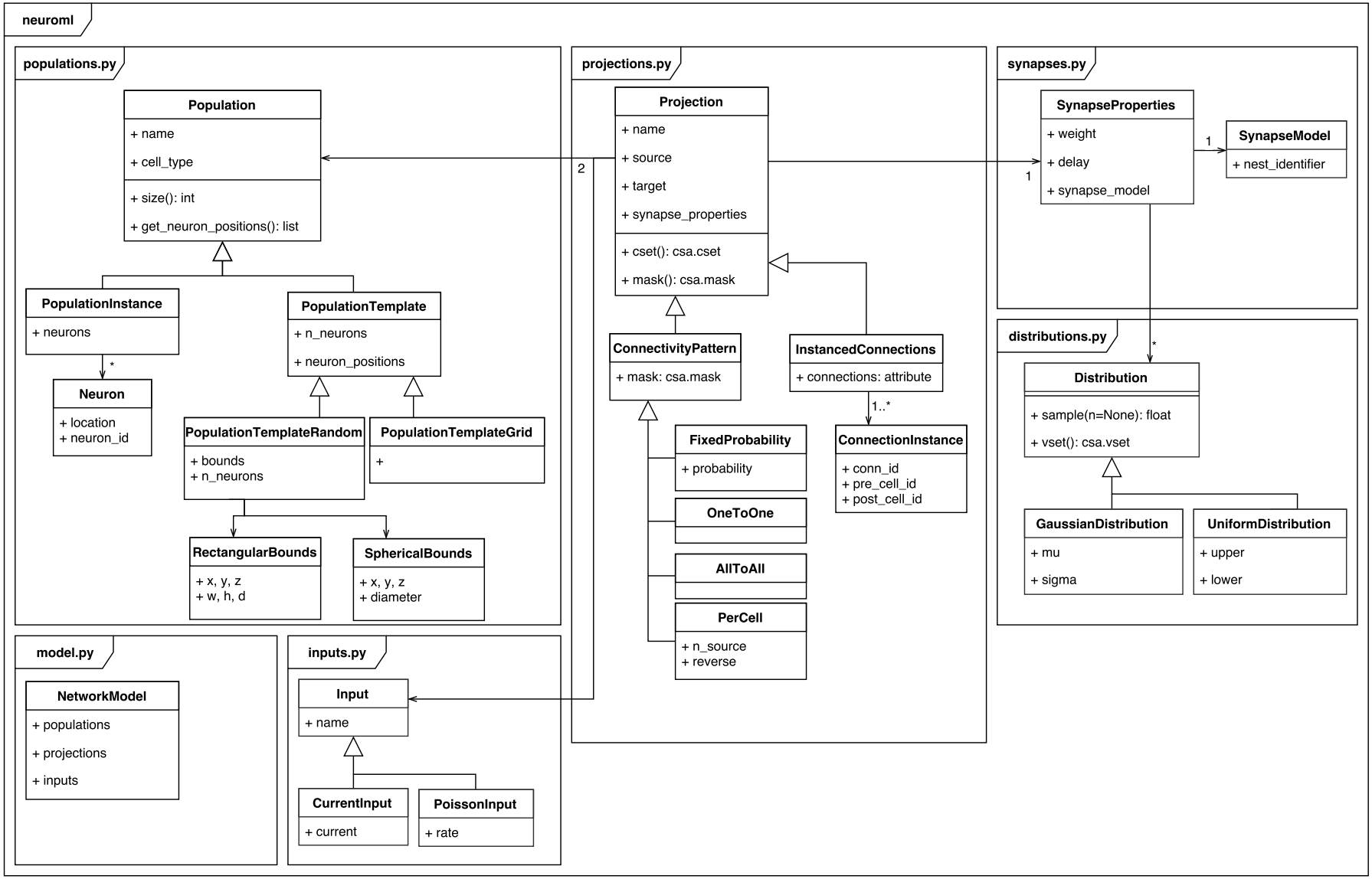


Figure 3.3: Class diagram of the `neuroml` package.

NetworkML tag	Python class	CSA structure
<all_to_all/>	AllToAll	csa.full
<one_to_one/>	OneToOne	csa.oneToOne
<fixed_probability/>	FixedProbability	csa.random(p)
<per_cell_connection/>	PerCell	csa.random(fanIn=n)
<gaussian_connectivity_2d/>	GaussianSpatialConnectivity	gaussian(sigma, cutoff)

Table 3.1: XML connectivity pattern tags and their corresponding Python classes and CSA structures. The Gaussian spatial connectivity has to be combined with CSA's random operator, which samples from the distribution.

3.2.3 XML Parsing

The NeuroML file is first read using the `xml.etree` module of the Python standard library. `xml.etree` is a standard python module for reading, modifying and writing XML files. Parsing an XML file with the module results in an `ElementTree` object, which can be traversed in a tree-like fashion. For increased speed, `xml.etree` provides a C implementation of `ElementTree` called `cElementTree`.

The module `vicogen.neuroml.parsing` then handles the parsing of NeuroML files for translation. There exists a parsing function for every XML tag that is defined by NeuroML. Most XML tags correspond to a Python class, as shown in Table 3.1. The parser first starts reading the populations, then the projections, and lastly the inputs. After the model has been parsed successfully, all string references between objects are replaced with object pointers. This allows later stages in the translation to easily follow the source and target of a projection.

The parser ignores tags it does not know, but if a known tag has wrong attributes or a tag is missing a required child tag, the parser throws a `NetworkMLParsingError`. This Python exception describes what went wrong and shows the offending XML block.

3.2.4 CSA Integration

To integrate the Connection Set Algebra into the class structure, the `ConnectivityPattern` class has a function `mask()` which returns a `csa.Mask` object. When `mask()` is not overridden by a subclass, it returns the class' `self._mask` attribute, which can be set by a subclass in the constructor. The translation process is a mapping of NeuroML tags to CSA objects. Table 3.1 shows the NeuroML representations and their assigned mask that defines the respective connectivity pattern.

The spatial connectivity can also be realized using CSA methods, but the translation process is more complex. For the distance metric the neuron positions have to be processed by the algebra. The neuron positions are either given by the neuron instance

elements in the NeuroML file, or by the position templates. If the positions are given by a template, the neuron positions are first sampled from the template's distribution. With the positions, a distance metric between the source and target neurons can be created.

```
pos_src = self.source.neuron_positions
pos_tar = self.target.neuron_positions
distance_metric = csa.euclidMetric2d(pos_src, pos_tar)
```

The distance metric is then combined with a Gaussian sampler. In the Python implementation of CSA, this is done by using the overloaded multiplication operator.

```
gaussian_metric = csa.gaussian(sigma, cutoff) * distance_metric
```

The Gaussian metric is then combined with a random operator. Note that in the Python implementation, `csa.random` can mean either a random mask, or a random operator, depending on the context. The random operator samples connections with the probabilities given by its other operand.

```
return csa.random * gaussian_metric
```

Value Sets

To transfer the synapse parameters such as synaptic weight and delay to the simulator, the parameters have to be expressed as value sets. Every projection has a set of parameters associated with them that have to be translated. A projection's parameter values are stored inside a `SynapseProperties` object. Since the Connection Generation Interface implementation in the `libneurosim` package only supports the synaptic properties *delay* and *weight*, only these two are implemented in `SynapseProperties`. When the simulator calls the connection generation library, the connection set is created from the connectivity pattern mask and the value set. The `cset()` function of `Projection` gathers the weight and delay parameters from the `SynapseProperties` object and creates a value set of them. This value set is combined with the mask to form a connection set. The connection set is then returned by the function, ready to be passed through the Connection Generation Interface.

If the connection is a connectivity pattern, the synapse properties may consist of distributed parameters instead of static parameters. Each distributed parameter needs a value set that represents the distribution. The `Distribution` class and its subclasses (so far only `UniformDistribution` and `GaussianDistribution`) provide implementations of the distributions added to NeuroML in Section 3.1.4. At runtime, the function `vset()` of the `Distribution` class creates a value set with a lambda function. This lambda function samples the distribution and is later called by the CGI.

3.2.5 Simulator Linking

The computational intensive part of the toolchain is the connection generation and simulation of the model. As a first proof of concept the NEST simulator is used because of its Python bindings and support for the CGI through the `libneurosim` package (See Section 1.3.2). While connectivity is managed in a simulator independent way, populations and inputs are not simulator independent. To be able to pass population and input information to the simulator, the `nest_.py` module provides functions to generate NEST populations out of a `Population` object. Inputs are handled in a similar way. This circumvents the need for a simulator independent interface for generating populations and input.

The `libneurosim` package provides the function `CGConnect()` to connect NEST populations using connection sets. The `CGConnect` function is also responsible for mapping the value sets to synapse parameters in NEST. Together with the two NEST populations and the connection set, the function also requires a Python dictionary that maps a string identifier of a parameter to the index of the corresponding value set. The translator only supports *weight* and *delay* as synapse parameters, as the `libneurosim` implementation of the connection generation interface only passes value sets with arity 0 or 2 to the simulator. For every projection, the translator calls:

```
nest.CGConnect(
    source_ids, # The neuron IDs of the source population
    target_ids, # The neuron IDs of the target population
    cset, # The connection set (with value set of arity 2)
    {
        'weight': 0, # The weight is stored in the first value set
        'delay': 1 # The delay is stored in the second value set
    })
```

This function queries all the connections that can be expressed by the connection set and connects the respective neurons in NEST.

To test the performance overhead of `CGConnect`, an alternative method was tested for performance: Instead of calling `CGConnect` for every projection, a single CSA mask was compiled of all projections and passed to the simulator. The single mask was created by converting neuron IDs to global IDs, and using the union operator to combine the individual masks. This method of generating connectivity showed a significant decrease in performance, due to the overhead the union operators add to the connectivity generation process. Keeping the masks simple is therefore more important than reducing the amount of `CGConnect` calls.

3.3 Extending `libcsa`

The C++ implementation of CSA offers increased performance over the Python implementation when generating connectivity. The `libcsa` library only supports a limited implementation of CSA, with only some elementary masks available. Masks can not be combined through set algebra in `libcsa`, and spatial connectivity is not available. Since the *fan in* and *fan out* connectivity that is part of NeuroML was not implemented in `libcsa`, the two missing connectivity patterns have been implemented.

3.4 Unit Tests

To make sure that the translator is working as intended during active development, unit testing has been implemented for the project. Unit tests allow for code validation and maintenance by automatically executing tests on the functionality of the code base. The tests can be run by executing `python python_unittest_runner.py` in the `test` directory.

All unit tests have been implemented using the Python framework `unittest`, which provides automated testing and testing functions. The tests are divided into three categories: NeuroML parsing, CSA translation, and NEST integration. The parsing test suite parses a set of predefined models form NeuroML and checks if the translator's internal object representation matches the model description. The test suite includes many different models, such as spatial models, models with parameter distributions and larger models. Testing the parser functions ensures that model information is not lost during the parsing process, and that the NeuroML standard is still supported. The CSA translation tests assure that the connectivity generated by CSA through the translator's functions are correct. When started, the test suite creates objects from the internal representation of NeuroML and translates them to CSA. All connectivity modes available in NeuroML are checked this way. Returned CSA objects are evaluated and checked against an existing connectivity matrix. The NEST integration tests ensure that CSA can properly communicate with NEST through the connection generation interface and that connectivity is generated correctly. Also tested are the functions for generating populations and inputs in NEST.

Chapter 4

Analysis

This chapter will present an analysis of the developed tools and visualizations, and if they meet the specified requirements.

4.1 Technical Analysis

Section 2.4 has introduced technical requirements for the translator. The developed translator is able to simulate a network from a NeuroML file, which fulfills the requirement T1. Furthermore, the translator is able to convert the connectivity of the model into the mathematical language of the Connection Set Algebra, and simulate the model using the Connection Generation Interface and the NEST simulator. Since the backend is written in Python, has a modular design, and uses the Connection Generation Interface, adding support for a new simulator (T2) is straight forward. As long as the simulator supports CGI, the simulator can be added to the translator by implementing the creation of populations and inputs, and the simulation control. The separation of frontend and backend (T3) has been fulfilled completely by using NeuroML files as an exchange point. The proposed additions to NetworkML (3.1) are necessary to use the translator to its full extend, although using standard NeuroML is still valid.

4.2 Performance

To ensure a performant scaling of the system with more workload and more data, the complexity of the algorithms used need to be scalable. Especially at the critical paths of the translation the efficiency of the code is important. Possible critical paths of the systems are the parsing of the NeuroML files, the translation and the connectivity generation.

Parsing the NeuroML files is bound by the speed of the XML parser. The best performance for parsing could be attained when using the C implementation provided

by the `xml.etree` Python module. The parsing functions all run in linear time as every object has to be parsed just once. The only part of the parsing process that does not run in linear time is the linking of projections to their populations. As the populations are identified by string in the XML, the two population objects with the correct string IDs have to be searched for every projection. For this the populations are saved in a Python dictionary for faster access by string. Theoretically, the population mapping has a time complexity of $O(n_{\text{pop}}n_{\text{proj}})$, where n is the number of populations or projections. Practically, since the number of projections and populations is usually low, the parsing has a very low impact on performance.

The translation is a simple mapping of NeuroML objects to CSA. When the connection generation is started, the translator creates the CSA objects for every projection. Since this is just the pattern and parameter definition, the performance of the translation is negligible.

The highest performance impact is the connectivity generation, which is bottlenecked by the connection generation library in use. When building the model with the simulator through the Connection Generation Interface, the connection generation library has to produce the actual neuron connections from connectivity patterns. Depending on the size of the network and the system hardware, the process may be very time consuming. The Python implementation of CSA is approximately 3.5 times slower than the C++ implementation, as shown in Figure 4.1. The performance of the C++ implementation is close to that of using NEST directly, proving that connectivity generation through the CGI with a minimal overhead is possible. Nevertheless, the algorithms of `libcsa` can be improved upon, as seen in the random connectivity generation in Figure 4.1. Since currently only the Python implementation of CSA can be used for translating a complete model, the technical requirement of performance (T4) was not met. Using `libcsa` as the connection generation library would increase performance drastically.

Usage on high performance computers is possible for the connection generation. CSA supports parallel execution through MPI. Since the connection generation is the bottleneck of the ViCoGen backend, parallel execution of the parsing and translation process is not needed. `libcsa` does not support parallel execution yet, but once it is fully implemented the technical requirement for parallelization (T5) can be considered as fulfilled.

4.2.1 Complexity Analysis

The algorithms used in CSA generally run in linear time, scaling with the amount of connections generated. The only algorithms running slower than linear time are the spatial connectivity masks. All algorithms for distributing neurons in space have to include an inverse function that finds the closest neuron to a point. This function is used during

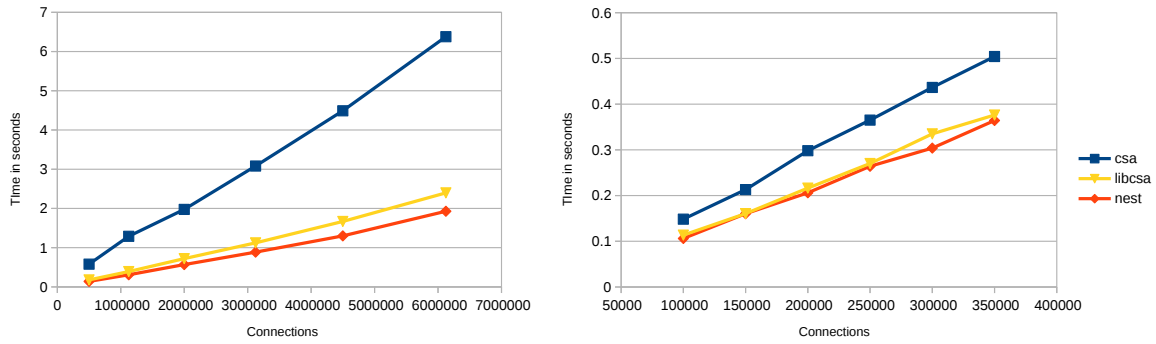


Figure 4.1: Connection generation performance for random connectivity (left) and one-to-one connectivity (right). Generation times were measured for the Python implementation (csa), the C++ implementation (libcsa), and generating connections directly in NEST. The computation is single threaded as the C++ implementation does not support multithreading. The Python implementation is about 3.5 times slower than using NEST directly. Using the C++ implementation shows some overhead, more noticeable when generating random connectivity. `libcsa` is generally 25% slower when generating random connectivity, and about 5% slower when generating one-to-one connectivity.

connection generation for spatial connectivity. The two- and three-dimensional random distributions implement the inverse search by iterating through every postsynaptic neuron for every presynaptic neuron to find the smallest distance. This results in a complexity of $O(n^2)$. A quadratic runtime can slow down the connectivity generation significantly. The performance of the inverse function could be improved by using a *KD-Tree* [10], which allows searching in a spatial structure in logarithmic time. Using a KD-Tree would make the complexity of the spatial connectivity $O(n \log n)$. A KD-Tree implementation for CSA was tested and a significant speedup was discovered for neuron populations greater than 10,000 neurons. For this, the KD-Tree implementation in the SciPy package [19] was used.

4.3 Usability Analysis

4.3.1 NeuroScheme Implementation

To fulfill the requirements for the GUI implementation, the team of the GMRV have worked on implementing the requirements for the GUI. They focused on the visual elements needed for displaying populations and projections, and implemented basic interactivity for all elements. For this, they have created a new domain in NeuroScheme, named *congen*. The *congen* domain allows users to visually design network models, with focus on the connectivity of the model. This section will examine the NeuroScheme *congen* domain and how the user stories from Section 3.1 have been implemented.

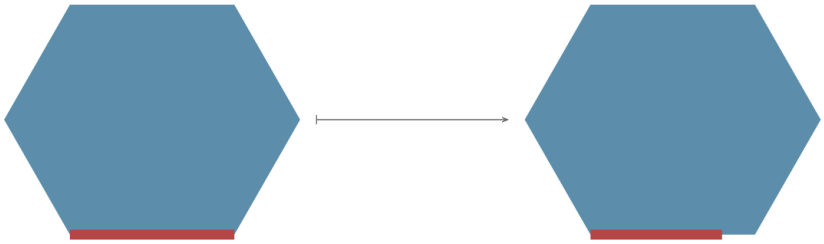


Figure 4.2: NeuroScheme's representation of connectivity as it is implemented so far. Populations are displayed by Hexagons, with a red bar indicating the relative population size normalized over all populations. The projection is visualized by an arrow connecting the two populations. The arrow's source is indicated by a bar. So far, arrow tip size representing the indegree has not been implemented. Future versions should look similar to Figure 2.2.

The visual language proposed in Section 2.2 has been mostly implemented. Populations are represented by a hexagonal shape with an indicator bar that shows the relative number of neurons in relation to other populations. The color of the hexagon relates to the neuron model used in the population. Projections are represented as arrows, with either a circle or an arrow tip at the target of the arrow, and a line at the arrow source to indicate the direction of the arrow. The arrow tip is used for excitatory connections, and the circle for inhibitory connections. At the start of the arrow is a perpendicular line, which makes discerning the direction of the arrow easier. The thickness of the arrow relates to the synaptic weight of the projection, and is scaled relative to other projection's weights. The size of the arrow tip should change based on the indegree of the projection, but has not been implemented yet. Hovering over a projection highlights it in red. When hovering over a population, the outgoing projections are highlighted in blue, and the incoming projections in orange.

The user stories focus on the interactivity with the system. Since NeuroScheme's interactivity features were built for examining data instead of creating data, the system is still missing features. The user stories have been implemented as follows:

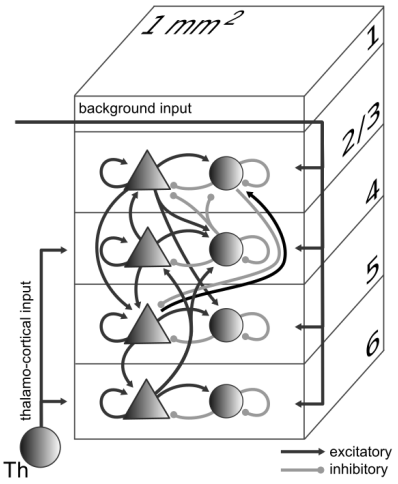
- Creating a neuron population (U1) is possible, but only through right clicking the scene. No buttons for the toolbar have been implemented yet. When *Add new neuron pop* is selected from the context menu, a dialog on the right hand panel appears, which allows the creation of multiple neuron populations at once.

- Creating projections between populations (U2) is possible, although still in need of better usability. When the *Display connectivity* mode is active (selectable in the toolbar), a projection can be clicked and dragged from one population to another. A dialog appears where the projections parameters can be set. While populations can be selected for editing, projections can not.

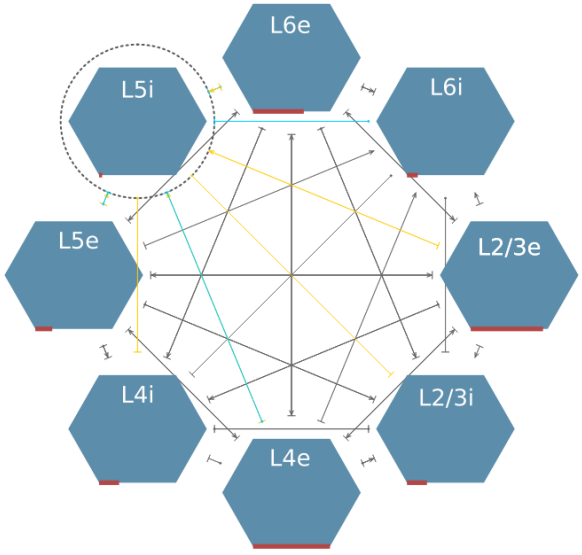
- Changing the projection parameters (U3) is only possible by creating a new projection over the old one. The old projection is then overwritten with the new parameters. Deleting projections is not possible.
- Creating and connecting inputs (U4) is not possible yet, as inputs do not have a representation in NeuroScheme yet.
- Searching for a specific element by name (U5) is not possible. The outline has not been implemented, but populations and projections can be given names. The name of a population can only be seen in the properties panel on the right when editing the population. Connection names can not be examined at all.
- Selecting multiple projections is also not possible yet, which makes editing of filtered selections (U6, U7) impossible.
- The grouping of elements (U8) has not been implemented. The populations can not be moved freely in the scene; only arranging populations in a grid or circular pattern is possible.
- The parameter and connectivity matrices (U9) have not been implemented.
- To simulate a created model (U10), the model has to be exported to a NeuroML file and then manually passed to the translator. Simulation directly from NeuroScheme is not possible. Exporting the model can be done by selecting *Scene* → *Save* in the menu bar.

4.3.2 Visual Language Analysis

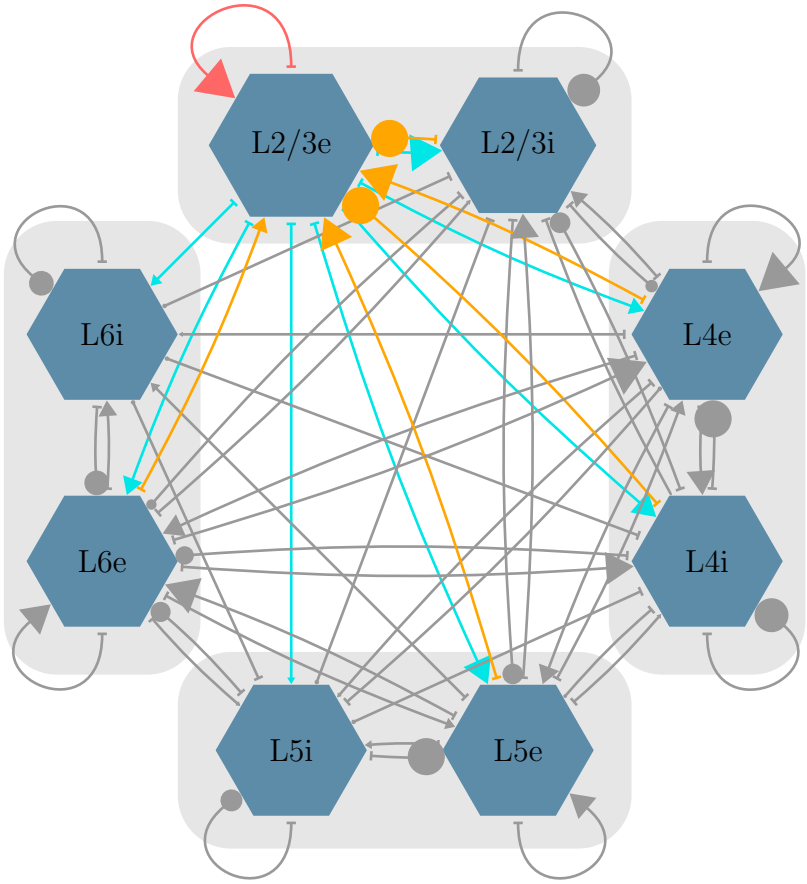
The visual language has been designed to be intuitive and expressive, and is based on interviews from neuroscientists and figures from neuroscience papers. If the visualization is actually useful for neuroscientists is still open for question. At the point of writing, the visual tool developed by the GMRV does not have enough functionality to test the usability to full extent. As a proof of concept, the model described in Potjans and Diesmann [26] (Fig. 4.3a) was implemented in the proposed language. The model of the cortical network described by Potjans and Diesmann consists of eight neuron populations, which are grouped into four layers with an inhibitory and excitatory population each. One thalamic population gives additional input to the layers. The model uses random connectivity, with probabilities compiled from anatomical and physiological data. The weights and delays of the models are distributed on a Gaussian distribution for all connections. Figure 4.3b shows what the model looks like when recreated with the current



(a) Original model from Potjans and Diesmann [26].



(b) Recreated model with current build of NeuroScheme. Labels have been added in for clarity.



(c) Concept design for recreated model

Figure 4.3: The original Potjans-Diesmann model (a), the recreated model in NeuroScheme (b), and how the model may look like in a later version of NeuroScheme (c).

implementation of the NeuroScheme tool. Since the visual language has not been fully implemented into NeuroScheme yet, it is missing some features (see 4.3.1). To fully show the capabilities of the visual language, the model has been recreated with PGF/TikZ in Figure 4.3c. The recreated visualization opens up a lot of details that may not be possible to deduct from the original representation:

- The high connectivity towards *L2/3e* is caused by the relatively large number of neurons, which is immediately visible through the arrow tips pointing at the layer.
- *L4e* has about the same number of neurons as *L2/3e*, but the arrows indicate a low connectivity towards *L4e*.
- The connections towards *L4e* are also mainly inhibitory, as indicated by the circular arrow tips.
- All layers have a high amount of recurrent connections.

These insights demonstrate the abilities of the visual language. The visual language is able to give information about the network which otherwise can only be attained through careful analysis of the connectivity matrix and population sizes.

Chapter 5

Conclusion and Future Work

This project has introduced a visual language for connectivity and a way of translating it to a simulator. The resulting ViCoGen package could be successfully tested with a cortical network model, simulating it from a simulator independent model description. Additionally, applying the visual language to the cortical network model provided a richer visualization for connectivity than existing representations.

To arrive at the visual language, usability interviews have been conducted and analyzed. Neuroscientists with different backgrounds were interviewed on their usage of connectivity in their models. Participants contributed their ideas for visual representations of connectivity, and provided insight into synaptic parameters. The interviews cleared up which parameters are important when modeling connectivity, and what visual representations are intuitive for neuroscientists.

Many projects have attempted to visualize the brain to allow a deeper and more intuitive understanding of connectivity. While 3D visualizations can show the spatial relation of brain regions, they lack the descriptive power of more abstract representations. NeuroScheme shows that the abstraction of visual elements can help the understanding of network models by only showing relevant information. The visual language for connectivity that has been developed based on the usability interviews tries to combine the visual abstractions of NeuroScheme, the intuitive representations used in neuroscience papers (e.g. Potjans and Diesmann [26]), and the information dense connectivity matrix visualizations used in Nordlie and Plesser [22].

With the resulting visual language, complex neuronal models could be recreated to gain a better insight into the connectivity of the model. As an implementation for the language, a GUI for building network models has been developed in collaboration with the GMRV in Madrid. NeuroScheme proves to be an adept framework for general visualizations in the neuroscience field, and its well modulated structure allows for implementing

the defined visual language. The interactivity of the system has been implemented by the GMRV, and basic models can be created and exported using the visual tool.

Furthermore, a way of translating the visual language has been achieved by combining various existing technologies into a translator. The translator fulfills most of the technical requirements stated in Section 2.4, and the critical path of translation can be successfully executed. Any model created with the GUI and the visual language can be expressed using the NeuroML standard and parsed by the translator, ensuring the modularity of the system. The translator has been developed to parse NeuroML files and passing them to the simulator. CSA has been applied successfully as a way of expressing the connectivity of the parsed models in a simulator independent way. To ensure a complete feature set of the translation, missing elements of CSA and NeuroML have been proposed and implemented. Through the Connection Generation Interface, the NEST simulator could be connected to the translator, allowing for the simulation of a model directly from the visual language.

While the requirements for performance could not be met with the Python implementation of CSA, the modular design allows exchanging the connection generation library with minimal development cost. The C++ implementation of CSA, `libcsa`, shows almost no overhead in the connection generation process in comparison to the simulator. Progress on extending `libcsa` has been done, and a complete version may be available soon. With a fully implemented `libcsa`, the performance requirement can be easily solved.

Simulator independent modeling has previously been attempted using PyNN as a simulator interface, but the problem of writing code remained. The ViCoGen toolchain allows the user to create a network model without writing simulation code. ViCoGen attempts to fill the gap between designing a model and simulating it by implementing the full process from the GUI to the simulator. Although ViCoGen will not replace writing simulation code entirely, the learning curve of using neuronal simulators can be greatly reduced, especially for fields outside of computer science or teaching.

The visual language provided is able to represent complex models in a way that rivals custom representations. Once the language is fully implemented in NeuroScheme, improving the visual language will be a continuous process as new applications for the GUI emerge. Making the visual language customizable by users may also be a useful feature, as one single representation is unlikely to cover all applications of the language. For this, the usability tests have to be expanded to more fields of neuroscience.

To improve the ViCoGen framework in the future, more work has to be put into extending the interfaces and improving performance. More simulators need to be added to the translator and to the Connection Generation Interface. So far only the NEST Sim-

ulator is fully supported by the translator. The problem of the Connection Generation Interface is that it can not separate the simulator from the translator completely. Populations, input and simulation control still lack a proper interface. Writing an interface for other parts of the simulation would improve the modularity of ViCoGen immensely, and adding new simulators would become easier.

The NeuroML framework works well for exchanging model information, but it could be used to its full extend in the future. The lower two levels of NeuroML, MorphML and ChannelML, could be used to provide the simulator with more detailed neuron and synapse models. Including all levels of NeuroML would allow more accurate small scale simulations, or even simulations of single neurons as biologically accurate as possible. On the side of the abstraction scale, the third level, NetworkML, may be improved to support more connectivity features. The connectivity patterns of NeuroML are limited to what is defined in NetworkML, with the exception of defining individual connections. More complex patterns like those in CSA can not be described in NeuroML. It may be possible to add the functionality of CSA to NeuroML, as both can be written in XML. If NeuroML would support a CSA connectivity pattern object, any possible pattern could be described using NeuroML.

To improve the performance of ViCoGen, the C++ implementation of CSA has to be developed to fully support the algebra. The performance of the system depends on the connection generation, thus having a performance optimized version is critical. `libcsa` also needs to support high performance computing and distributed systems, as large scale networks require significant computing power. So far, only a small subset of the Connection Set Algebra is implemented in `libcsa`, but a rewrite of the codebase may be necessary to support all features.

The project opens up many additional complementary projects. For example, the results of a simulation have to be passed back to the GUI, processed, and displayed. Such a project would include finding appropriate visualizations of simulation results, interfaces to extract the data from simulators, and data processing tools. Further projects are the control and *in situ* visualization of a simulation during run time, comparing results between simulation, and raw data storage. All these projects are outlined in the *Modular Science Framework*, proposed by Wouter Klijn and Sandra Diaz. The Modular Science framework aims to provide tools for all areas of computational network models. All the parts of the framework are built in a modular way, so that researchers can plug their own tools into the framework through a set of interfaces, allowing them to use the visualization, analysis and simulation tools provided. The modules are controlled through Module Orchestration software, which allows the selection of modules, the communication between them, and smaller tasks such as logging and monitoring. The framework would

allow faster development of new tools and more efficient reuse of existing tools. The ViCoGen project denotes the foundation of this Modular Science framework, and is the first step towards a more complex and extensive system.

Acknowledgment

I would like to thank the team at the GMRV, especially Dr. Pablo Toharia Rabasco and Sergio E. Galindo Ruedas, for their work and support on NeuroScheme; Mikael Djurfeldt for his contribution and support on CSA and `libcsa`; and the interview participants for taking their time to help me get this project along. Huge thanks goes to Sandra Diaz Pier and Wouter Klijn at the Forschungszentrum Jülich for their amazing support during the creation of this thesis. Not only have I learned a lot from them about the neuroscience field and scientific working, but they were always ready to provide generous feedback and support. I could not have wished for better supervisors on this project.

Chapter 6

Supplements

6.1 Example Models

Spatially Connected Model

```
<neuroml xmlns="http://morphml.org/neuroml/schema">
    <populations xmlns="http://morphml.org/networkml/schema">
        <population name="PopA" cell_type="iaf_psc_alpha">
            <pop_location>
                <random_arrangement population_size="75">
                    <rectangular_location>
                        <corner x="0" y="0" z="0"/>
                        <size width="2" height="3" depth="1"/>
                    </rectangular_location>
                </random_arrangement>
            </pop_location>
        </population>
        <population name="PopB" cell_type="iaf_psc_alpha">
            <pop_location>
                <random_arrangement population_size="50">
                    <rectangular_location>
                        <corner x="0" y="0" z="0"/>
                        <size width="0.5" height="2.5" depth="2"/>
                    </rectangular_location>
                </random_arrangement>
            </pop_location>
        </population>
    </populations>

    <inputs>
        <input name="InputA">
            <random_stim frequency="600"/>
        </input>
    </inputs>

    <projections units="Physiological Units" xmlns="http://morphml.org/networkml/schema">
        <projection name="PopA-PopB" source="PopA" target="PopB">
            <synapse_props synapse_type="StaticSynapse" threshold="-20">
                <weight> 90.0 </weight>
```

```
            <internal_delay> 0.5 </internal_delay>
        </synapse_props>
        <connectivity_pattern>
            <gaussian_connectivity_2d sigma="1.5" cutoff="3"/>
        </connectivity_pattern>
    </projection>
    <projection name="InputA-PopA" source="InputA" target="PopA">
        <synapse_props synapse_type="StaticSynapse" threshold="-20">
            <weight> 90.0 </weight>
            <internal_delay> 0.5 </internal_delay>
        </synapse_props>
        <connectivity_pattern>
            <all_to_all/>
        </connectivity_pattern>
    </projection>
</projections>
</neuroml>
```

Distributed Parameter Model

```
<neuroml xmlns="http://morphml.org/neuroml/schema">
    <populations xmlns="http://morphml.org/networkml/schema">
        <population name="PopA" cell_type="iaf_psc_alpha"/>
        <population name="PopB" cell_type="iaf_psc_alpha"/>
    </populations>

    <projections units="Physiological Units" xmlns="http://morphml.org/networkml/schema">
        <projection name="Gaussian" source="PopA" target="PopB">
            <synapse_props synapse_type="StaticSynapse" threshold="-20">
                <weight>
                    <GaussianDistribution center="87.8" deviation="8.8"/>
                </weight>
                <internal_delay>
                    <GaussianDistribution center="3.5" deviation="0.75"/>
                </internal_delay>
            </synapse_props>
            <connectivity_pattern>
                <all_to_all/>
            </connectivity_pattern>
        </projection>
        <projection name="Uniform" source="PopA" target="PopB">
            <synapse_props synapse_type="StaticSynapse" threshold="-20">
                <weight>
                    <UniformDistribution lower="1" upper="2.5"/>
                </weight>
                <internal_delay>
                    <UniformDistribution lower="0.1" upper="0.75"/>
                </internal_delay>
            </synapse_props>
            <connectivity_pattern>
                <all_to_all/>
            </connectivity_pattern>
        </projection>
    </projections>
</neuroml>
```

6.2 Installation Instructions

6.2.1 CSA and NEST with CGI

1. Install autoconf using `sudo apt-get install autoconf`
2. Clone and install libneurosim:

```
git clone https://github.com/INCF/libneurosim.git
cd libneurosim
./autogen.sh
./configure --prefix=$HOME/opt/libneurosim
make
make install
```

3. Install csa:

```
git clone https://github.com/INCF/csa.git
cd csa
./autogen.sh
./configure --with-libneurosim=$HOME/opt/libneurosim
make
sudo make install
```

4. Install NEST:

```
git clone https://github.com/nest/nest-simulator
cd nest-simulator
cmake .. -Dwith-libneurosim=$HOME/opt/libneurosim
make
sudo make install
```

Used Versions

- libneurosim at commit acc5b1a6445c82d66fba9bb8681360e00d7a3c9d
- csa at commit a5cc33a56549cf76656dfab0af135bc55f7d1cda
- NEST at commit efd79cb486d299b08cf0796dc5d2cc1ade7581fb
- autoconf 2.69-9
- Python 2.7.12

6.2.2 Compiling and installing libcsa

In the `libcsa` root, execute:

```
bash autogen.sh
./configure CXXFLAGS="-std=c++14"
make
sudo make install
```

After that, you have to add the path to the `LD_LIBRARY_PATH` environment variable for your terminal:

```
export LD_LIBRARY_PATH=/usr/local/lib
```

You can add this to your `.bashrc` file to execute always on bash startup. You may have to add this variable to any IDE you are using.

6.2.3 Installing NeuroScheme

1. Clone from Git (e.g. `git clone git@gitlab.gmrv.es:nsviz/NeuroScheme.git`)
2. Install Boost using `sudo apt-get install libboost-all-dev`
3. Install Qt
   (a) Download an installer from `https://www.qt.io/download-open-source/`
   (b) If using the online installer, you have to make it executable using `chmod +x qt-unified-linux-x64-2.0.5-2-online.run`
   (c) Run the installer
4. Download and extract Eigen 3 from `http://eigen.tuxfamily.org`
   - Inside the eigen directory, make a new build directory, build and install using these commands:

     ```
     mkdir build_dir
     cd build_dir
     cmake ..
     make
     sudo make install
     ```

5. To build NeuroScheme, run the following commands inside the NeuroScheme source folder:

```
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make rebase
make
sudo make install
```

6. This will install NeuroScheme into `/usr/bin/NeuroScheme`. Running `sudo make install` is optional. You can also run NeuroScheme directly from `build/bin/NeuroScheme`.

   (a) cmake will fail if the folders `FiReS`, `scoop`, and `ShiFT` are not inside the NeuroScheme root folder, and are not downloaded automatically during cmake. Download the repositories from a different source and place the three folders inside the NeuroScheme source root.

   (b) Rerun `cmake .. -DCMAKE_BUILD_TYPE=Release`, then `make` and `sudo make install`.

When pulling a new version, make sure to use `make rebase` to update all dependencies from their git repositories.

Starting ConGen

To start the interactive connectivity generation, use `./NeuroScheme -d congen`. Right click the workspace to add new neuron populations. Left click and drag to create a new connection. Make sure *Show connectivity* is active (top icon bar) to see connections. To save a model, select *Scene* → *Save* from the menu bar.

Used Versions

- ubuntu-16.04.2-desktop-amd64
- NeuroScheme at commit `66f56823cc841e38d4c09f0f7b29744b513a79c0`
- scoop at commit `cc02a23db899f06a6cfa9c17d41b795f5e9971be`
- FiReS at commit `0da27f916ec54d69d4aff32be5b253fab496ec4a`
- ShiFT at commit `97ec57cda8481c47dd81eae480aa1ccf0a56ed4b`
- Boost version 1.58.0.1ubuntu1
- Qt 5.2 (qt-unified-linux-x64-2.0.5-2)
- Eigen 3.3.3

6.2.4 Installing ViCoGen

The Python package does not need to be installed to be used, but can be added to your Python libraries. In the ViCoGen root, execute:

```
mkdir build && cd build
cmake ..
make
sudo make install
```

6.3 ViCoGen Usage Instructions

ViCoGen can be used independently of the GUI. The ViCoGen package can be imported in Python using `import vicogen`. Alternatively, ViCoGen can be used directly from the console by executing it as a Python module in the command line.

```
python -m vicogen [-h] [-o [OUTFILE]] [-t SIMTIME] [-c]
                  [--nest-options NEST_OPTIONS] [-d] [-v]
                  modelfile
```

`modelfile` is the NeuroML file to be parsed and has to be given. The module supports the following command line arguments:

Argument	**Description**
`-h, --help`	Shows a help message on the usage.
`-o [OUTFILE], --outfile [OUTFILE]`	Set the file to write output to. If not given, the output is written to `stdout`.
`-t SIMTIME, --simulate SIMTIME`	Simulate the model for a given amount of milliseconds using NEST.
`-c, --write-connections`	Instead of simulating the network, parse the connections and write them to output.
`--nest-options NEST_OPTIONS`	Additional options for NEST.
`-d, --debug`	Print debugging information.
`-v, --verbose`	Print verbose messages.

Bibliography

[1] James A. Bednar. Topographica: Building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components. *Frontiers in Neuroinformatics*, 3:8, 2009. URL http://dx.doi.org/10.3389/neuro.11.008.2009.

[2] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, March 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2010.118.

[3] JM Bower and D Beeman. The book of genesis: exploring realistic neural models with the general neural simulation system. 1998. *Telos, Springer, New York*.

[4] Robert C. Cannon, Marc-Oliver Gewaltig, Padraig Gleeson, Upinder S. Bhalla, Hugo Cornelis, Michael L. Hines, Fredrick W. Howell, Eilif Muller, Joel R. Stiles, Stefan Wils, and Erik De Schutter. Interoperability of neuroscience modeling software: Current status and future directions. *Neuroinformatics*, 5(2):127–138, Apr 2007. ISSN 1559-0089. doi: 10.1007/s12021-007-0004-5. URL https://doi.org/10.1007/s12021-007-0004-5.

[5] Robert C. Cannon, Padraig Gleeson, Sharon Crook, Gautham Ganapathy, Boris Marin, Eugenio Piasini, and R. Angus Silver. Lems: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning neuroml 2. *Frontiers in Neuroinformatics*, 8:79, 2014. ISSN 1662-5196. doi: 10.3389/fninf.2014.00079. URL https://www.frontiersin.org/article/10.3389/fninf.2014.00079.

[6] Andrew Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. Pynn: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2:11, 2009. ISSN 1662-5196. doi: 10.3389/neuro.11.011.2008. URL https://www.frontiersin.org/article/10.3389/neuro.11.011.2008.

[7] Mikael Djurfeldt. The connection-set algebra—a novel formalism for the representation of connectivity structure in neuronal network models. *Neuroinformatics*, 10 (3):287–304, Jul 2012. ISSN 1559-0089. doi: 10.1007/s12021-012-9146-1. URL https://doi.org/10.1007/s12021-012-9146-1.

[8] Mikael Djurfeldt, Andrew P. Davison, and Jochen M. Eppler. Efficient generation of connectivity in neuronal networks from simulator-independent descriptions. *Frontiers in Neuroinformatics*, 8:43, 2014. ISSN 1662-5196. doi: 10.3389/fninf.2014.00043. URL https://www.frontiersin.org/article/10.3389/fninf.2014.00043.

[9] Jochen M Eppler, Moritz Helias, Eilif Muller, Markus Diesmann, and Marc-Oliver Gewaltig. PyNEST: a convenient interface to the NEST simulator. *Frontiers in Neuroinformatics*, 2(12), 2009. ISSN 1662-5196. doi: 10.3389/neuro.11.012.2008.

[10] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3 (3):209–226, September 1977. ISSN 0098-3500. doi: 10.1145/355744.355745. URL http://doi.acm.org/10.1145/355744.355745.

[11] Wulfram Gerstner and Werner M. Kistler. *Spiking neuron models : single neurons, populations, plasticity / Wulfram Gerstner ; Werner M. Kistler.* Cambridge [u.a.] Cambridge Univ. Press 2008, 2008. ISBN 9780521890793. URL http://widgets.ebscohost.com/prod/customerspecific/s9118275/vpn.php?url=http://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=cat04752a&AN=rub.2737758&lang=de&site=eds-live&scope=site.

[12] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[13] Padraig Gleeson, Volker Steuber, and R. Angus Silver. neuroconstruct: A tool for modeling networks of neurons in 3d space. *Neuron*, 54(2):219 – 235, 2007. ISSN 0896-6273. doi: https://doi.org/10.1016/j.neuron.2007.03.025. URL http://www.sciencedirect.com/science/article/pii/S0896627307002486.

[14] Padraig Gleeson, Sharon Crook, Robert C. Cannon, Michael L. Hines, Guy O. Billings, Matteo Farinella, Thomas M. Morse, Andrew P. Davison, Subhasis Ray, Upinder S. Bhalla, Simon R. Barnes, Yoana D. Dimitrova, and R. Angus Silver. Neuroml: A language for describing data driven models of neurons and networks with a high degree of biological detail. *PLOS Computational Biology*, 6(6):1–19, 06 2010. doi: 10.1371/journal.pcbi.1000815. URL https://doi.org/10.1371/journal.pcbi.1000815.

[15] Dan Goodman and Romain Brette. The brian simulator. *Frontiers in Neuroscience*, 3:26, 2009. ISSN 1662-453X. doi: 10.3389/neuro.01.026.2009. URL https://www.frontiersin.org/article/10.3389/neuro.01.026.2009.

[16] Moritz Helias, Susanne Kunkel, Gen Masumoto, Jun Igarashi, Jochen Eppler, Shin Ishii, Tomoki Fukai, Abigail Morrison, and Markus Diesmann. Supercomputers ready for use as discovery machines for neuroscience. *Frontiers in Neuroinformatics*, 6:26, 2012. ISSN 1662-5196. doi: 10.3389/fninf.2012.00026. URL https://www.frontiersin.org/article/10.3389/fninf.2012.00026.

[17] M. L. Hines and N. T. Carnevale. The NEURON simulation environment. *Neural Computation*, 9(6):1179–1209, Aug 1997.

[18] David H Hubel. *Eye, brain, and vision.* New York, NY, US: Scientific American Library/Scientific American Books, 1995.

[19] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL http://www.scipy.org/.

[20] Susanne Kunkel, Abigail Morrison, Philipp Weidel, Jochen Martin Eppler, Ankur Sinha, Wolfram Schenck, Maximilian Schmidt, Stine Brekke Vennemo, Jakob Jordan, Alexander Peyser, Dimitri Plotnikov, Steffen Graber, Tanguy Fardet, Dennis Terhorst, Håkon Mørk, Guido Trensch, Alex Seeholzer, Rajalekshmi Deepu, Jan Hahne, Inga Blundell, Tammo Ippen, Jannis Schuecker, Hannah Bos, Sandra Diaz, Espen Hagen, Sepehr Mahmoudian, Claudia Bachmann, Mikkel Elle Lepperød, Oliver Breitwieser, Bruno Golosio, Hendrik Rothe, Hesam Setareh, Mikael Djurfeldt, Till Schumann, Alexey Shusharin, Jesús Garrido, Eilif Benjamin Muller, Arjun Rao, Juan Hernando Vieites, and Hans Ekkehard Plesser. Nest 2.12.0, March 2017. URL https://doi.org/10.5281/zenodo.259534.

[21] Henry Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

[22] Eilen Nordlie and Hans Ekkehard Plesser. Visualizing neuronal network connectivity with connectivity pattern tables. *Frontiers in Neuroinformatics*, 3:39, 2010. ISSN 1662-5196. doi: 10.3389/neuro.11.039.2009. URL https://www.frontiersin.org/article/10.3389/neuro.11.039.2009.

[23] Eilen Nordlie, Marc-Oliver Gewaltig, and Hans Ekkehard Plesser. Towards reproducible descriptions of neuronal network models. *PLOS Computational Biology*, 5

(8):1–18, 08 2009. doi: 10.1371/journal.pcbi.1000456. URL https://doi.org/10.1371/journal.pcbi.1000456.

[24] C. Nowke, M. Schmidt, S. J. van Albada, J. M. Eppler, R. Bakker, M. Diesmann, B. Hentschel, and T. Kuhlen. Visnest – interactive analysis of neural activity data. In *2013 IEEE Symposium on Biological Data Visualization (BioVis)*, pages 65–72, Oct 2013. doi: 10.1109/BioVis.2013.6664348.

[25] Luis Pastor, Susana Mata, Pablo Toharia, Sofia Bayona, Juan Pedro Brito, and Juan Jose Garcia-Cantero. NeuroScheme: Efficient Multiscale Representations for the Visual Exploration of Morphological Data in the Human Brain Neocortex. In Mateu Sbert and Jorge Lopez-Moreno, editors, *Spanish Computer Graphics Conference (CEIG)*. The Eurographics Association, 2015. doi: 10.2312/ceig.20151208.

[26] T. C. Potjans and M. Diesmann. The cell-type specific connectivity of the local cortical network explains prominent features of neuronal activity. *ArXiv preprint arXiv:1106.5678*, June 2011. URL https://arxiv.org/abs/1106.5678.

[27] Ivan Raikov, Robert Cannon, Robert Clewley, Hugo Cornelis, Andrew Davison, Erik De Schutter, Mikael Djurfeldt, Padraig Gleeson, Anatoli Gorchetchnikov, Hans Ekkehard Plesser, Sean Hill, Mike Hines, Birgit Kriener, Yann Le Franc, Chung-Chan Lo, Abigail Morrison, Eilif Muller, Subhasis Ray, Lars Schwabe, and Botond Szatmary. Nineml: the network interchange for neuroscience modeling language. *BMC Neuroscience*, 12(1):P330, Jul 2011. ISSN 1471-2202. doi: 10.1186/1471-2202-12-S1-P330. URL https://doi.org/10.1186/1471-2202-12-S1-P330.

[28] Olaf Sporns, Giulio Tononi, and Rolf Kötter. The human connectome: A structural description of the human brain. *PLOS Computational Biology*, 1(4), 09 2005. doi: 10.1371/journal.pcbi.0010042. URL https://doi.org/10.1371/journal.pcbi.0010042.

[29] Ruben A. Tikidji-Hamburyan, Vikram Narayana, Zeki Bozkus, and Tarek A. El-Ghazawi. Software for brain network simulations: A comparative study. *Frontiers in Neuroinformatics*, 11:46, 2017. ISSN 1662-5196. doi: 10.3389/fninf.2017.00046. URL https://www.frontiersin.org/article/10.3389/fninf.2017.00046.

[30] Mingrui Xia, Jinhui Wang, and Yong He. Brainnet viewer: A network visualization tool for human brain connectomics. *PLOS ONE*, 8(7):1–15, 07 2013. doi: 10.1371/journal.pone.0068910. URL https://doi.org/10.1371/journal.pone.0068910.

Erklärung

Ich erkläre, dass das Thema dieser Arbeit nicht identisch ist mit dem Thema einer von mir bereits für eine andere Prüfung eingereichten Arbeit.
Ich erkläre weiterhin, dass ich die Arbeit nicht an einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen der Arbeit, die anderen Werken mit dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen der Entlehnung kenntlich gemacht. Dies gilt sinngemäß auch für gelieferte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Datum

Unterschrift