Faculty for Physics and Astronomy

University of Heidelberg

Bachelor thesis

in Physics

submitted by

Alexander Müller-Brand

born in Wiesbaden

August 2010

Testing the Accuracy of Neuromorphic Device Configurations

This bachelor thesis has been carried out by

Alexander Müller-Brand

at the

Kirchhoff Institute for Physics
University of Heidelberg

under the supervision of

Prof. Dr. Karlheinz Meier

Testing the Accuracy of Neuromorphic Device Configurations

This thesis originates from the core problem that as complexity in neuromorphic modeling rises, the ability to spot flaws in the model implementation and configuration, especially in case of recurrent neuronal networks, diminishes strongly. This problem is approached by providing methods that ensure the validity, functionality, and a maximum degree of accuracy of neuronal network model implementations. To this end, various high-level neuronal network tests have been developed, which check the correct mapping of neuronal network descriptions to hardware-specific configurations. These tests are integrated into a newly developed framework, which has been specifically designed towards flexibility in incorporating complex and heterogeneous testing workflows. Experimental proof of the versatility, applicability, and benefits of the high-level neuronal network tests is presented. The tests are used to check the functionality of two state-of-the-art hardware back-ends developed within the community of the FACETS research collaboration.

Verfahren zum Testen der Akkuraten Konfiguration Neuromorpher Hardwaresysteme

Ausgangspunkt dieser Arbeit ist ein Kernproblem des neuromorphen Modellierens, nämlich dass die Fehleridentifikation und -suche in der Implementation und Konfiguration eines Modells, besonders im Falle von rekurrenten neuronalen Netzwerken, durch die vorhandene Komplexität stark erschwert wird. Ein Lösunganansatz für dieses Problem sind die hier präsentierten Methoden, welche die Validität, Funktionalität und ein Höchstmaß an Genauigkeit einer neuronalen Netzwerkimplementation sicherstellen sollen. Zu diesem Zweck sind diverse Tests auf Basis funktionaler Mikronetzwerke entwickelt worden, welche die richtige Übersetzung von Beschreibungen neuronaler Architekturen zu der entsprechenden hardware-spezifischen Konfiguration überprüfen. Diese Tests werden in eine neu entwickelte Softwarestruktur integriert, welche besonders auf Flexibilität bei der Einbindung von komplexen und heterogenen Testmodulen ausgelegt ist. Die Vielseitigkeit, Anwendbarkeit, und Vorteile der Mikronetzwerke als Testeinheiten werden experimentell aufgezeigt. Insbesondere werden die Tests abschliessend auf zwei aktiv in der Forschung eingesetzten Hardwareimplementationen demonstriert, welch im Rahmen der Forschungsgemeinschaft FACETS entwickelt wurden.

Contents

Introduction **1**

1. Materials and Methods **3**

1.1. Neuroscience Basics . . . 3
1.1.1. Biological Neurons . . . 3
1.1.2. Neuron Model Dynamics . . . 5
1.2. Used Software and Hardware . . . 8
1.2.1. The FACETS Stage 1 Hardware . . . 9
1.2.2. The FACETS Stage 2 Hardware . . . 10
1.2.3. Various Software Simulation Packages . . . 12
1.2.4. The Simulator-Independent Modeling Language PyNN . . . 12
1.3. Testing . . . 17
1.3.1. Testing Methods . . . 17
1.3.2. Testing Levels . . . 19

2. Comprehensive Test Framework **21**

2.1. The Test Framework Inspector . . . 21
2.1.1. Requirements for the Framework . . . 22
2.1.2. Choice of Language . . . 22
2.1.3. The Interface between the Framework and Tests . . . 23
2.1.4. The Design of the Framework . . . 24
2.2. High-Level Network Tests . . . 25
2.2.1. Requirements . . . 26
2.2.2. Constructing a Versatile Test Architecture . . . 26
2.2.3. Statistical Considerations . . . 30
2.2.4. The Test Cases: Varying Constituents and Connections . . . 31
2.2.5. Integration of Tests into the Test Framework . . . 33

3. Application of Tests 35
3.1. Testing NEST and NEURON 35
3.2. Testing the FACETS Stage 1 Hardware 39
3.3. Testing with the FACETS Stage 2 Executable System Specification 42

Conclusion and Outlook 46

A. Resources and Supplements 49
A.1. Source Code of the Test Framework 49
A.2. Source Code of the High-Level Neuronal Tests 49
A.3. Using the Test Framework 49

Bibliography 51

Introduction

The cerebral cortex is the outermost layer of the mammalian brain. In humans, it is only a few millimeters thick, yet its emergence in mammals represents a significant step in the evolution of life on our planet, as it plays a fundamental role in learning, cognition, consciousness, and intelligent behavior. Its further development in the human brain contributed to the dominant role humans have taken on our planet, and gave rise to many of mankind's greatest inventions and discoveries ranging from art and language to mathematics and science, including neuroscience, which investigates this very neural tissue at its core. It is the unprecedented size of the cortex in human brains which enables us to study ourselves and the inner workings of our brain and separates us from the animal kingdom.

There is, however, a fundamental problem neuroscience faces, and that is the complexity of the interconnected and vastly intricate cell structures found in the human brain. The human brain is estimated to contain 10 to 100 billion neurons, which pass signals to each other via as many as 100 - 1000 trillion synaptic connections [52]. Because our brain is so complicated in structure we have to find a level of abstraction, which enables us to experimentally and quantitatively model our brain (see e.g. [7, 44] for two reviews on modeling strategies), and allows us to focus on what really *is* important.

This, however, is a not an easy thing to do because the fundamental understanding of what mechanisms are essential for thought processes is still very narrow. Hence, many approaches in neuroscience attempt to incorporate as many details as possible, and consequently and necessarily drag along a lot of this complexity. But as complexity rises, the ability to spot mistakes and flaws in the implementation of a model diminishes. This fact represents the core problem, which has been addressed in the thesis at hand. It is essential to develop methods ensuring the validity, functionality, and a maximum degree of accuracy of any neuronal network model implementation.

In the Electronic Vision(s) group at the Kirchhoff-Institute for Physics in Heidelberg[1], where this thesis has been written, different approaches to modeling are studied with a particular focus on hardware implementations [47, 48, 49, 50]. Hardware, while offering numerous advantages such as scalability and speed, has, however, its own issues that

[1] http://www.kip.uni-heidelberg.de/vision/

need to be taken into account. Due to the difficulties faced in the manufacturing process of integrated circuits [12, Section 6.5.3], imperfections and artifacts cannot be avoided completely. This creates an additional layer of potential problems besides those found in the software stack used to configure and access the hardware [9, 8, 17]. Furthermore, due to the wide spectrum in the dynamics of active recurrent neuronal networks, identifying errors within these dynamics is hard.

With this in mind thorough testing of all components is a fundamental necessity, especially the more complex a neuronal network implementation becomes as it tries to get closer to biological reality.

In the modeling approach of the Electronic Vision(s) group, many components are utilized to form a complete implementation of a neuromorphic modeling platform, which is novel in its speed, size, and configurability. Tests are necessary which check the correct functionality of the individual components as well as that of the complete system. The presented work provides a solution to and framework for testing the interplay of all modules and demonstrates their successful application.

This thesis is structured as follows. Chapter 1 introduces the reader to the basics of neuroscience, the used hardware and software, and contemporary testing methods. Chapter 2 discusses the development of a test framework and individual high-level neuronal network tests, which have been developed for this thesis. Chapter 3 shows the application of the high-level neuronal network tests to software simulators and hardware implementations provided by the Electronic Vision(s) group. Chapter 4 finishes by providing a summary, conclusion and an outlook to further research. Additional information and references to the source code can be found in the Appendix.

1. Materials and Methods

1.1. Neuroscience Basics

This section introduces the reader to the basics of neuroscience with a focus on the biological background and neuron model dynamics.

1.1.1. Biological Neurons

Over the last decades, research in the fields of biology and neuroscience has given scientists great insight into the structure and functionality of the brain and its most fundamental functional components. Already around the year 1900 Santiago Ramón y Cajal, a Spanish histologist, psychologist, and Nobel laureate [43], observed that the brain is, at a low level, made up of cells called *neurons*, which act as information processing units, with often long, branching connections in between them [20]. There are also other types of cells in the human brain besides spiking neurons, which also appear to play an essential role in its functionality (see e.g. [15] for a classification of cortical neuron types), however, as they are not relevant for this thesis, they are not discussed here any further.

A neuron can be divided into three main components: dendrites, soma, and axon. The *soma*, or cell body, which contains the cell nucleus, can be described as the central core of the neuron, which integrates information and plays a significant role in information processing. This information reaches the soma via *dendrites*, which are connected to the soma and extend away from it with often many branches. The complete set of dendrites is often called a *dendritic tree*. While dendrites can best be described as a carrier for information to the soma, *axons* can be described as a carrier for information away from the soma (while also being able to carry feedback into it). The axon is a slender prolongation of the soma, which can extend up to tens of thousands of times the diameter of the soma in length. Most neurons have only one axon, but this axon usually undergoes extensive branching, thus enabling communication with many target cells. A single neuron in the mammalian cortex often connects to more than 10^4 neurons [4]. The bridge between the axon of one neuron and the dendrites of another neuron is called a *synapse*. The *presynaptic*

neuron is the cell whose axon is part of the synaptic link, which lies on a dendrite of the *postsynaptic neuron*.[1]

The voltage difference between the interior and the exterior of the neuron is referred to as the *membrane potential*. At rest, the potential lies at about -65 mV (see e.g. [31, 6, 55, 51]for models with corresponding parameter value assumptions), which is called the *resting potential*. Information is delivered to the soma in the form of so-called *postsynaptic potentials* (henceforth abbreviated as PSP), voltage changes, which travel along dendrites towards the soma. If the change in voltage is positive, the synapse which provoked the PSP is referred to as an *excitatory synapse* and the potential is referred to as an *excitatory postsynaptic potential*, or EPSP. In case the change is negative, the synapse is said to be *inhibitory* and the potential to be an *inhibitory postsynaptic potential*, or IPSP. It has been observed that, in general, at all axonal branches of a single neuron, the same set of neurotransmitters is released, making neurons either purely excitatory or purely inhibitory.

Multiple PSPs are subject to summation until they reach a certain *threshold voltage* at the soma, which causes the generation of a so-called *action potential* or *spike* at the axon hillock. It is generally assumed that, in the mammalian cortex, the exact shape of action potentials carries less information than their number and timing [20, 45].

Spikes are much higher in magnitude than the sum of all the incoming PSPs. Action potentials generated by a soma are in the order of about 100 mV and have a typical duration of about 1 to 2 ms, while PSPs created by synapses have an amplitude of about 1 mV, with a width of up to several tens of milliseconds. Since the firing threshold typically lies 10 to 20 mV above the resting potential, it is necessary for multiple PSPs to arrive at the soma at about the same time in order to reach the threshold value and trigger a single action potential. Shortly after the action potential has been sent out by the soma, the membrane potential falls below the resting potential, initiating the *absolute refractory period*. Within the absolute refractory period of a neuron it is impossible for a neuron to generate another spike. After the absolute refractory period, a period of *relative refractoriness* takes over, in which it is possible but difficult to generate a second spike [20].

As a spike travels along an axon, it will eventually hit the synaptic bridge between the axon and dendrites of the target neuron.

In a *chemical synapse*, which is the most common type of synapse in the mammal brain [15], the presynaptic and postsynaptic neurons are separated by a gap about 20 nm wide referred to as the *synaptic cleft*. When a spike reaches the synapse it triggers the release of a chemical substance called a *neurotransmitter* into the synaptic cleft. The

[1] While most synapses are connections between axons and dendrites, there are also other types of connections, including axon-to-soma, axon-to-axon, and dendrite-to-dendrite [15].

neurotransmitter binds to receptors residing in the postsynaptic end of the synaptic cleft. There they cause the opening of different types of ion channels, so that ions can diffuse through the cell membrane changing its potential.

An excitatory synapse will trigger a PSP with positive amplitude (depolarizing EPSP), while an inhibitory synapse will trigger a negative amplitude PSP (hyperpolarizing IPSP). These PSPs now travel along the dendrites of the postsynaptic cell heading towards the soma of the target neuron. The quantity of neurotransmitters released into the synaptic cleft and the effectiveness with which the postsynaptic end responds to those neurotransmitters determines the strength or *weight* of a synaptic connection. The ability of the synapse to change its weight is often referred to as *synaptic plasticity* and is believed to play a crucial role in memory and learning [24, 36, 34, 35].

In an *electrical synapse* the presynaptic and postsynaptic cell membranes are connected by channels that are capable of passing electrical current, causing voltage changes in the presynaptic cell to induce voltage changes in the postsynaptic cell, essentially allowing direct electrical coupling between two neurons. This makes electrical synapses faster and more reliable than chemical synapses, even though the latter are more common [15].

1.1.2. Neuron Model Dynamics

In order to simulate the behavior characteristics of neurons, some of which have been outlined in Section 1.1.1, scientists have developed various mathematical models that simplify the more complex behavior of real neurons. Two of these models are used primarily by the back-ends tested during this thesis, which is why they require further explanation.

The Integrate-and-Fire Model

In a simple model, the so-called Integrate-and-Fire model, which was first researched in the context of neuroscience in 1907 by Lapicque [1], a neuron is described as a capacitor. For a capacitor it holds true that

$$Q = CV \tag{1.1}$$

where Q is the charge of the capacitor, and V the voltage between its ends, which represents the membrane potential of the neuron.

The first derivative with respect to time yields the current charging the capacitor

$$I(t) = C\frac{dV}{dt} \tag{1.2}$$

When an external current is applied, the voltage increases or decreases with time. If the voltage reaches a certain (constant) threshold voltage V_T, a spike is triggered. After the spike is sent out, the voltage is reset to the resting potential. After that, the model continues from the start. The frequency with which spikes can be generated thus increases linearly with respect to the external current without any limitations.

This unlimited spike frequency is not realistic behavior, and the model can be made more accurate by implementing the absolute refractory period t_{ref} (see Section 1.1.1) after a spike has occurred. During this period no second spike can be triggered. This essentially limits the firing frequency of a neuron. The firing frequency thus obtained as a function of a constant input current is

$$f(I) = \frac{I}{CV_T + t_{ref}I} \tag{1.3}$$

Another problem with this simplified model is that it has no time-dependent memory. In case a current charges the capacitor but is not strong enough to trigger a spike, the capacitor will retain that voltage increase forever until it fires. This means that if a certain number of incoming spikes are required to arrive close in time in order to trigger an outgoing spike in a real neuron, this model would allow them to arrive with practically unlimited distance in time. This is clearly not in line with observed neuronal behavior.

This remaining shortcoming can be overcome by introducing a "leakage" in the form of a resistor, which is in parallel with and constantly drains the capacitor [30]. With this additional element, when spikes are too far apart in time, their voltage boost decreases over time until the membrane potential reaches the rest potential.

Now the current can be split into two components, namely the resistive current I_R going through the resistor and the capacitive current I_C charging the capacitor:

$$I = I_C + I_R \tag{1.4}$$

Ohm's law gives the resistive current:

$$I_R = \frac{V}{R} = g_L \cdot V \tag{1.5}$$

with V being the voltage across the resistor, R being the resistance of the resistor, and g_L being the conductance of the resistor. This leads to

$$I = C \cdot \frac{dV}{dt} + g_L \cdot V \tag{1.6}$$

Now the membrane time constant τ_m is introduced with

$$\tau_m = \frac{C}{g_L} \tag{1.7}$$

and Equation 1.6 can be written as

$$\tau_m \cdot \frac{dV}{dt} = -V + \frac{I}{g_L} \tag{1.8}$$

Immediately after the creation of a spike the membrane voltage drops to a new value called the reset potential from where it rises again back to the rest potential. This is not apparent from the equations presented and is indeed an artificial constraint introduced to make the model more realistic.

Looking at Equation 1.6 shows however, that the capacitor is leaking towards a rest voltage of 0. To formulate the equation in a more realistic manner one can embed the so-called leak reversal potential, which is hereby denoted as V_L and leads to Equation 1.10.

$$I = C \cdot \frac{dV}{dt} + g_L \cdot (V - V_L) \tag{1.9}$$

This causes the leakage to drain the capacitor towards the rest potential, V_L.

The external current I is made up of positive current created by ion channels of excitatory synapses and negative current created by ion channels of inhibitory synapses. This is reflected by differential Equation 1.10, which allows calculating the temporal behavior of the membrane potential.

$$\sum_E I_E(V,t) + \sum_I I_I(V,t) + g_L \cdot (V - V_L) = -C \cdot \frac{dV}{dt} \tag{1.10}$$

The two sums are over excitatory and inhibitory synapses, denoted by indices E and I, respectively. Each type of ion channel has its own reversal potential as can be seen in Equations 1.11 and 1.12.

$$I_E(V,t) = p_E(t) \cdot w_E(t) \cdot g_E^{max}(t) \cdot (V - V_E) \tag{1.11}$$

$$I_I(V,t) = p_I(t) \cdot w_I(t) \cdot g_I^{max}(t) \cdot (V - V_I) \tag{1.12}$$

The maximum conductance of a synapse is denoted as $g_S^{max}(t)$, the synaptic weight as $w_S(t)$, and $p_S(t)$ can be interpreted as the percentage of open ion channels in the synapse. V_E and V_I are the excitatory and inhibitory reversal potentials. One can also introduce time constants τ_{syn_E} and τ_{syn_I} that characterize the decline of the conductance of excitatory and inhibitory synapses, respectively.

Because the Integrate-and-Fire model has such a high degree of simplification compared to the intricate biological reality, it offers only limited spike-timing prediction power. Nevertheless, it is often used for modeling neurons that do not exhibit adaptive behavior.

The Adaptive-Exponential Integrate-and-Fire Model

The *Adaptive-Exponential Integrate-and-Fire* model [6], or short *AdEx*, adds several additional mechanisms to the Integrate-and-Fire model (see Section 1.1.2) and is described by Equations 1.13 and 1.14:

$$C \cdot \frac{dV}{dt} = -g_L \cdot (V - V_L) + g_L \Delta_T \exp(\frac{V - V_T}{\Delta_T}) - w + I \tag{1.13}$$

$$\tau_w \frac{dw}{dt} = a(V - V_L) - w \tag{1.14}$$

V denotes the membrane potential, V_L the leak reversal potential, V_T the threshold potential, I the input current, C the membrane capacitance, g_L the leak conductance, Δ_T the so-called slope factor describing the sharpness of spikes, a the adaptation coupling parameter, w the adaption variable, and τ_w is the adaptation time constant.

Equation 1.13 describes the temporal evolution of the membrane potential, while Equation 1.14 describes the temporal evolution of the adaption current w.

The exponential function in Equation 1.13 models the voltage-dependent initiation of an action potential. In the mathematical interpretation of the equations, the action potential diverges towards infinity. Integration of Equations 1.13 and 1.14 is, however, usually stopped at a high, finite potential (usually above 0 mV) and at that time the spike is said to occur. The decline of the action potential is not modeled by the equations. In this model it is simplified by a reset of the voltage to the rest potential at firing time. At this time the adaption value is also shifted by a certain amount, introducing a spike-triggered adaptation.

The two-dimensional AdEx model is able to reproduce all common firing patterns found in biological neurons [37], and can be very accurately fitted to biological data [27].

1.2. Used Software and Hardware

The following subsections give an overview over the used software and hardware. In the following the term back-end refers to the simulator, either in software or hardware form, which implements given neuron and synapse models to simulate neuronal networks.

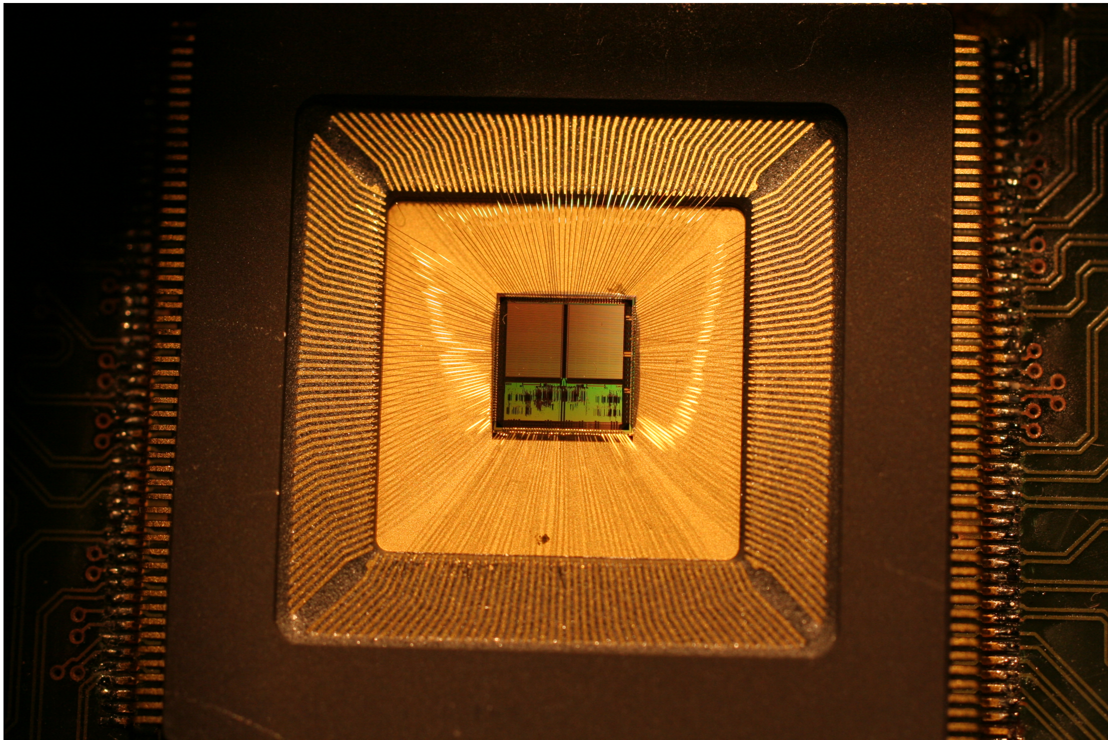

Figure 1.1.: A photograph of the Spikey chip.

1.2.1. The FACETS Stage 1 Hardware

This section intends to provide the reader with relevant details about the FACETS *Stage 1* hardware, often referred to as *Spikey*. Spikey is a neuromorphic prototype chip developed by the Electronic Vision(s) group and has been described in detail in various publications [47, 48, 8, 29, 10] developed by the Electronic Vision(s) group. Figure 1.1 shows an actual photograph of a Spikey chip taken with a microscope. The chip is realized in a 0.18 µm CMOS[2] technology.

A Spikey chip hosts 384 neurons which use a conductance-based leaky Integrate-and-Fire model (see section 1.1.2). Each neuron is fed by up to 256 synapses and a total of 100000 synapses are supported. The Spikey chip implements spike-time-dependent plasticity and is highly accelerated with respect to biological real time and provides an acceleration factor of 10^4. That is, a neuronal network on a Spikey chip would process neuronal events up to 10^4 faster than its biological counterpart. This acceleration is part of the design and arises from the very-large-scale integration (VLSI) of electronic circuits and their intrinsically short time constants.

[2]Complementary Metal-Oxide-Semiconductor

As no two neurons are the same in the real mammal brain, so are transistors and electronic parts different as a direct result of fluctuations and imperfections in the manufacturing process. This is demonstrated in Section 3.2, where high-level neuronal network tests are run on Spikey chips.

1.2.2. The FACETS Stage 2 Hardware

This section introduces the reader to the FACETS *Stage 2* hardware [49, 16, 50], the core of which is a *wafer* containing nearly 400 neuromorphic chips. This wafer-scale system will be present in the near future providing up to 200K neurons and 50 million synapses (see Figure 1.2 for a rendered computer model of the system), but as of the time this thesis has been written is only existent in the form of a Executable System Specification [56].

The wafer will be made up of a number of reticles. Each reticle contains eight so-called *HICANN* chips, which stands for *High Input Count Analog Neuronal Network*. Wafer-scale connections are used to link HICANNs together. HICANNs provide the bred and butter of a neuronal network, namely neurons and synapses, which implement the ADEX neuron model (see Section 1.1.2) to model neuron dynamics and the same plasticity mechanisms as the FACETS Stage 1 hardware, also working with a speedup factor of 10^4. Each HICANN is able to host a maximum of 512 neurons and 128 thousand synapses, adjacent neuron circuits can be interconnected such that a neuron can receive up to 16K synaptic inputs. Spike transmission from neurons to synapses located on any HICANN on the wafer is performed by the so-called *Layer 1* network: a dense but highly configurable grid of horizontal an vertical bus lanes connect all HICANNs with each other, spikes are transmitted via asynchronous serial digital pulse packets representing the ID of the sending neuron. The synchronous *Layer 2* communication is responsible for routing pulses to neurons on the wafer (e.g. for stimulation) and for routing recorded pulse events to the host computer or to neurons on other wafers.

Digital network chips, or *DNC* for short, provide communication bandwidth for all HICANNs of a reticle, which are connected to one common DNC using Layer 2 communication channels. Additionally a bidirectional connection to a *Field Programmable Gate Array*, or *FPGA* for short, is provided by the DNC.

The FPGA acts as the central interface for communication between the wafer and the host and between individual wafers. Four DNCs are connected to a single FPGA. This implies that one FPGA has to deal with the configuration and routing data, and neuron events of up to 16 thousand neurons.

This short introduction into the make-up of the Stage 2 hardware already reveals the

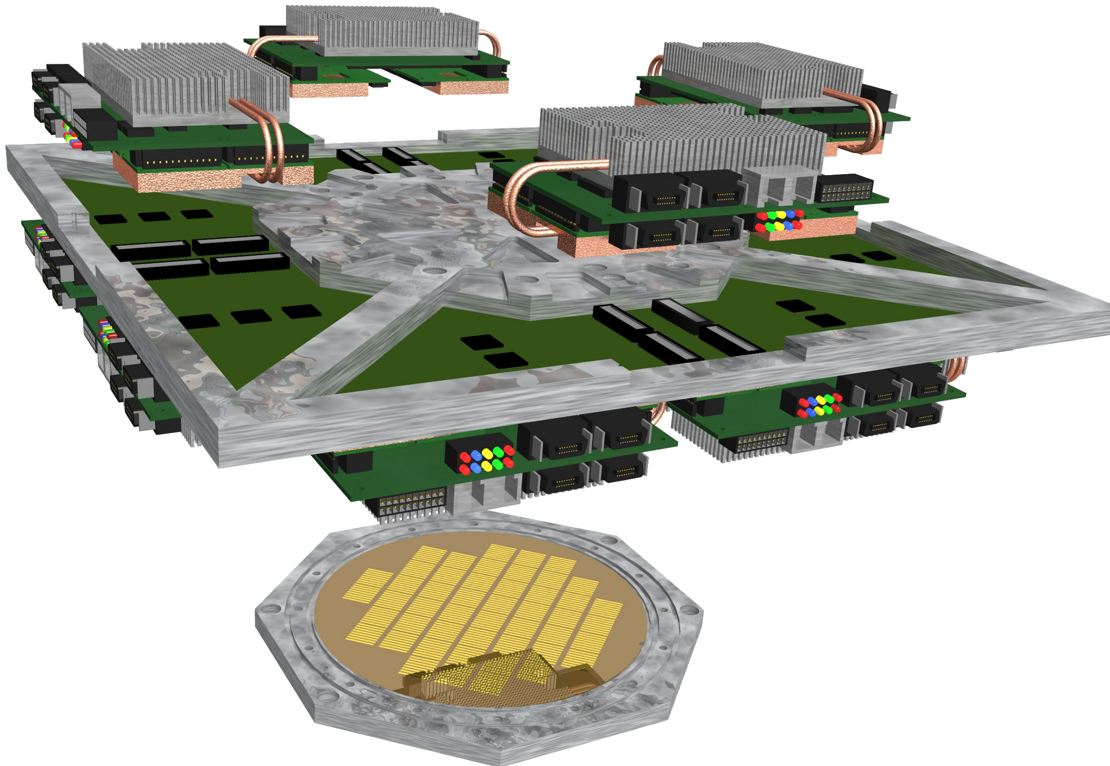

Figure 1.2.: A rendered computer model of the FACETS Stage 2 wafer-scale hardware system. Figure by D. Husmann.

great complexity that comes with such a comprehensive system. There are multiple sources of error, such as an erroneous mapping of neuronal models onto the wafer or hardware failures. The great variety in configuration scenarios and manifold ways of signal routing impose an even greater challenge as the correct realization of a biological neuron architecture is paramount. In order to cope with these problems in an efficient and elegant manner, generic, high-level neuronal network tests have been developed during this thesis, which are introduced in Section 2.2. These tests also serve the purpose of checking the correct functionality of the Executable System Specification of the Stage 2 hardware and have already helped in finding problems in the software operating the hardware (see Section 3.3).

1.2.3. Various Software Simulation Packages

The software simulators *NEST*[3] [14, 21, 18] and *NEURON*[4] [25, 11] are used for this thesis as a back-end for checking the validity of the high-level, neuronal network tests (see Section 2.2 and 3.1). Compared to hardware back-ends, software is very flexible as it allows the implementation of additional neuron models.

NEURON was originally developed by John W. Moore at Duke University and has since been greatly improved and benefited from a large user base, efficient implementation of neuron model dynamics, and comprehensive documentation and maintenance. It allows to simulate very sophisticated neuron and synapse models, which include the spatial structure of the cell, and implements the Integrate-and-Fire (see Section 1.1.2) and Adaptive-Exponential Integrate-and-Fire (see Section 1.1.2) models.

NEST, which is short for *Neural Simulation Technology*, is a simulator similar to NEURON and is capable of handling large heterogeneous neuronal networks. NEST advantages become apparent when using models that focus on architecture of neuronal systems rather than on the detailed morphological and biophysical properties of individual neurons. NEST, too, implements the Integrate-and-Fire (see Section 1.1.2) and Adaptive-Exponential Integrate-and-Fire (see Section 1.1.2) models.

PCSIM [39, 40] and *Brian* [22] are further examples for commonly applied software simulators, but they have not been used for this thesis.

1.2.4. The Simulator-Independent Modeling Language PyNN

This section introduces the reader to the simulator-independent modeling language *PyNN* [13, 41], which is used for this thesis and throughout the Electronic Vision(s) group. Furthermore, the *mapping process* is described that is needed to transfer a neuronal network defined with PyNN to the specific neuromorphic back-ends developed in that group.

Motivation

As apparent from Sections 1.2.1, 1.2.2, and 1.2.3, there is a variety of different back-ends, each with its own interface and configuration process. If one wants to simulate the same neuronal network on different back-ends (e.g. for benchmarking or for the verification that a network's behavior is not only an artifact of a certain back-end, or because a newer back-end provides more performance), its realization has to be reprogrammed and reconfigured

[3] http://www.nest-initiative.org
[4] http://www.neuron.yale.edu

from scratch for every new back-end. This leads to an additional workload that decreases the efficiency of the development process.

It is hence desirable to have a common interface which remains stable even when the implementation changes. Simulators and hardware may come up or lose ground, but the general concept of what a neuron or synapse is remains the same if we stick to the same model. So it makes sense to have an interface which describes agreed-upon terms, while at the same time hiding unnecessary detail and implementation-specific characteristics.

For example, in the Electronic Vision(s) group software simulators like NEST or NEURON (see Section 1.2.3) are used together with hardware platforms like the already mentioned Stage 1 and Stage 2 systems, with the wafer-scale version of the latter still in development as of the time this thesis has been written. It would be cumbersome if the very same network needed to be rewritten specifically for each of these platforms. Especially for the case of hardware, which suffers from manufacturing imperfections and requires a customized calibration, reference software simulations of the same neuronal experiments are of great value. It quickly becomes apparent that there is great motivation for an abstract network-describing language, which hides implementation details. This is where PyNN provides a solution.

Python as a Glue Language

PyNN uses *Python* [42, 46, 33], a general-purpose, high-level programming language, the design philosophy of which emphasizes code readability. It is an interactive, object-oriented, and interpreted programming language and provides high-level data structures such as tuples, lists, and associative arrays, dynamic binding and dynamic typing, modules, classes, and exceptions. It has a relatively simple syntax, yet is a powerful language due to its general purpose character and versatility. As many other scripting languages it is free of cost, even when used for commercial purposes, and it is capable of running on practically any modern computer. A Python program, when run, is compiled into platform independent byte code by the interpreter, which is then interpreted.

The term *glue language* usually describes a computer programming language whose purpose is to connect different software components together. Python lends itself to be used as a glue language, because it spans multiple platforms, it is easy to learn and to maintain, and provides a high-level interface to the system as a whole. For this reason, Python is a good candidate for a back-end-independent modeling language like PyNN.

The Interface of PyNN

PyNN [13] acts as a simulator- and hardware-independent language for describing and building neuronal network models. It is based on Python. Writing tests with PyNN allows to test multiple simulators with only minor modifications to the code. PyNN takes care of translating the neuron, synapse and network models into the required configuration for a given simulator, consistent handling of physical units, and consistent handling of random number generation, and provides an object-oriented, high-level interface to easily enable structured development of large-scale, complex models.

This allows for testing simulators as well as constructing neuronal networks without worrying about the underlying implementation. As of the time this thesis has been written the NEURON, NEST, PCSIM, and Brian software simulators and the FACETS hardware systems are supported [9]. As PyNN hides implementation details, tests written in PyNN are good for black-box-testing 1.3.1 underlying platforms. An additional characteristic of PyNN is that its execution involves multiple layers: all the way from creating an abstract model of the neuronal network to generating a configuration representing that model in the actual back-end. This may be seen as an advantage, because one can test multiple levels with a single test. It may, however, also be seen as an disadvantage, because once a flaw has been detected, it requires additional work to determine at which level the error occurred.

The API[5] can be roughly divided into two components: a low-level, procedural API, allowing the creation of neurons and synapses at a low level with functions like `create()`, `connect()`, `set()`, `record()`, `record_v()`, and a high-level, object-oriented API allowing the creation of whole populations of neurons with classes like `Population` and `Projection`, which have methods like `set()`, `record()`, `setWeights()`.

The low-level API lends itself for small networks and provides more flexibility than the high-level API. The high-level API is good for hiding details, allowing the developer to concentrate on the overall structure of one's neuronal model.

PyNN also translates standard cell model names and parameter names into simulator- or hardware-specific names, which is good for platform-independence. For example, standard model `IF_curr_alpha` is called `iaf_neuron` in NEST and `StandardIF` in NEURON, while `SpikeSourcePoisson` is called a `poisson_generator` in NEST and a `NetStim` in NEURON.

[5]Application Programming Interface

Creating a PyNN script in Python can be as simple as

```
# Import PyNN.
import pyNN.nest as pynn
# Set up PyNN.
pynn.setup()
# Create stimulus.
stimulus = pynn.create(pynn.SpikeSourcePoisson, {'rate' :  100.  ,
'duration' :  1000}, n=1)
# Create neuron.
neuron = pynn.create(pynn.IF_cond_exp, n=1)
# Connect stimulus to neuron.
pynn.connect(source=stimulus, target=neuron, weight=0.015,
synapse_type='excitatory')
# Record spikes produced by neuron to a file.
pynn.record(neuron, 'recorded-spikes.txt')
# Run simulation.
pynn.run(1000)
# Shut PyNN down.
pynn.end()
```

The first line of this example shows an **import** call, which imports the simulator- or hardware-specific implementation of the PyNN API. That is usually the only place where one *has* to deal with platform-specific commands.

The next call to **pynn.setup()** starts PyNN. It allows for the specification of optional, back-end-specific parameters, that are used by a given simulator but not by others.

Following the setup call are commands which construct the neuronal network.

After the construction of the neuronal network **pynn.run()** is called, with the first parameter specifying the biological simulation time in milliseconds. The run-call blocks and instructs the underlying platform to actually simulate the neuronal network. It returns once this is complete.

Eventually PyNN is shut down by a call to **pynn.end()**, which stores recorded spike times and membrane voltages in files and cleans up memory.

As one can already see from this very short example, the only step needed to run the neural network on a different platform is to swap the import statement with the appropriate one for the target platform.

The Mapping Process

While neuronal networks built with the description language PyNN are theoretically unlimited in size and complexity, hardware back-ends have only a limited configuration space. Hence descriptions of a network most often cannot be *mapped* correctly to the available hardware. Loss of neurons or synapses can occur during this mapping process, which makes optimizing the process desirable and often necessary.

There are three main steps executed during mapping (see Figure 1.3): mapping of neurons (hereinafter simply called *placement*), mapping of synapses (hereinafter called *routing*), and mapping of parameter values (the so-called *parameter transformation*).

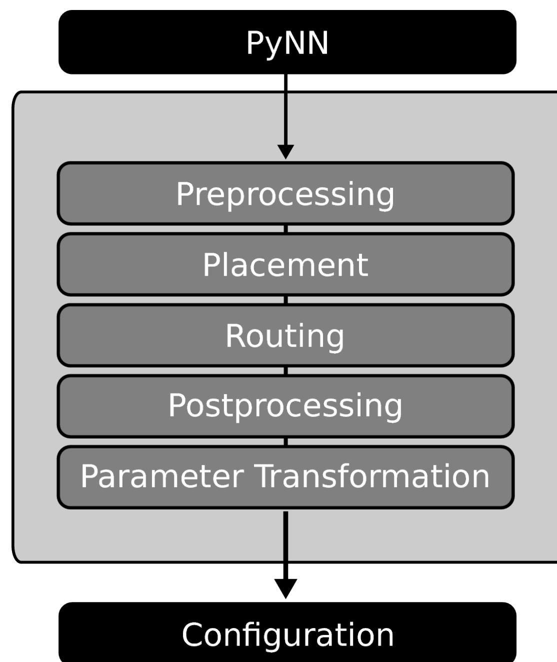

Figure 1.3.: A schematic view of the mapping process.

An abstract data model that represents both the biological architecture of the to-be-mapped neuronal network and the hardware configuration has been developed for this specific purpose: it is called *GraphModel* [58, 57]. The GraphModel is a hierarchical hypergraph, which contains vertices and edges, that hold data objects and relationships between data objects, respectively. The major advantage of this abstract and overarching data model is its generic nature resulting in efficiency in the development process of mapping algorithms. Both intermediate and final results are stored using the GraphModel, including the data needed for placement, routing, and parameter transformation.

Besides preprocessing and postprocessing, the first step taken by the mapping process is

the placement of neurons. A correct and optimized placement is important as it determines the overall performance and accuracy of the final hardware configuration with respect to the biological network. The mapping process attempts to utilize the available hardware space in the best manner possible, while minimizing the routing distances between interconnected neurons.

Connection routing is the proper translation of synaptic links into corresponding hardware connections [19]. It is followed by the proper transformation of biological parameter ranges to their hardware counterparts. Special care needs to be taken as the realizable value ranges on hardware are limited by such factors as the bit sizes used for parameter storage.

For a more thorough summary of the mapping process see [17].

Problems in the mapping process can arise in many ways. There is often no trivial solution that correctly maps the neuronal network description to an actual hardware configuration, sometimes leading to missing synaptic connections on the hardware system. Additionally, the limitations in the number of neurons and parameter ranges may be exceeded. Parameter values, while continuous in PyNN, are often discrete in hardware. Hardware failures are also possible, as well as coding mistakes in the software stack. The potential for problems is huge and it is therefore important to test the complex functionality of the mapping process as a whole. A solution for this is provided by the high-level neuronal network tests introduced in Section 2.2.

1.3. Testing

Testing models and algorithms, especially if they reside in software form and before recreating them in hardware, is essential to making sure that the observed behavior matches its specifications and expectations, so that they are fit for use.

1.3.1. Testing Methods

There are mainly three different approaches to software testing. These three approaches are used to describe the point of view that a test engineer takes when designing test cases.

With *white-box testing* the tester has access to the actual implementation and internal structure of the code (see Figure 1.4). There are several types of white box testing to be named. There is *API testing*, which is the testing of the application using public and private, documented API calls and checking their behavior against their specifications. Then there is code coverage, which involves creating tests to satisfy some criteria of *code coverage* (e.g. all statements in the program to be executed at least once). Additionally,

test engineers may utilize fault injection methods, which improves the coverage of a test by introducing faults to test code paths. Mutation testing methods involve modifying source code of programs in small ways and considering any tests that pass after code has been mutated to be defective. Finally, there is static testing, the manual or automatic review of source code without actually compiling and running it.

The second approach is *black-box testing*. Black-box testing, as opposed to white-box testing, treats a component as a black box, that is, with a defined interface but without any knowledge whatsoever about its implementation and internal structure (see Figure 1.5).

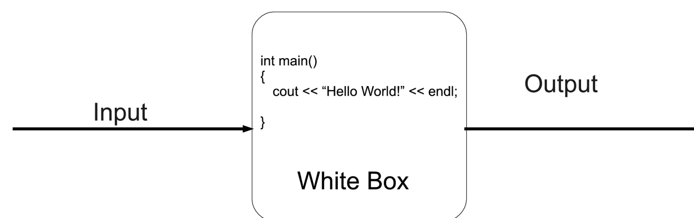

Figure 1.4.: White-box testing schema. The module to be tested is represented by the box, while input and output patterns are generated and evaluated, respectively, by the test. Detailed information about the inner workings of the module itself can be involved in this process.

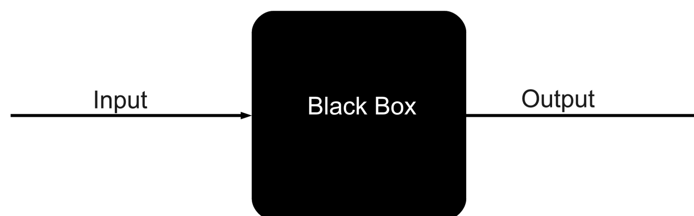

Figure 1.5.: Black-box testing schema. The module to be tested is represented by the box, while input and output patterns are generated and evaluated, respectively, by the test. No information about the module except of its input and output interfaces are involved in this testing process.

Its advantage is that, given that the test developer and developer of the code to be tested are two different people or groups of people, the test writers do not subliminally make assumptions about the implementation. A good example of this subliminally influence

is that of a code developer, who implements an algorithm, of which he has a flawed understanding, and then uses this flawed understanding to also develop test cases. This example demonstrates that it is generally a good idea to assign black-box testing to different developers.

Another advantage of black-box testing is that the implementation can change arbitrarily; as long as the interface stays the same, the tests remain valid and need not be modified. This decreases the maintenance costs of black-box tests and enhances their efficiency.

Gray-box testing is a mixture between black-box testing and white-box testing and involves performing tests with knowledge of internal implementation details like data structures and algorithms for purposes of designing the test cases, but testing at the interface level like a black-box test. It is essentially in between black box and white box testing, hence the name gray box testing. It combines both advantages and disadvantages of black- and white-box testing.

The main testing method used for this thesis is black-box and gray-box testing. As described in Section 2.2, neuronal network tests have been conceived, which do not rely on any specific back-end and hence are, by design, black-box tests. They "know", at least in principle, nothing about the back-end or its implementation. However, some care has been taken while choosing the neuron and stimuli parameters in order not to hit any upper or lower limits inherent to the implementation tested. This step makes the black-box tests become partly gray-box tests.

1.3.2. Testing Levels

Besides the aforementioned testing methods there are also different testing levels.

Unit testing, also called component testing, refers to dividing code up into smaller components, usually at the function level or, with object oriented programming languages, at the class level, and testing each function or class separately [5, 23]. One class or function might have multiple and diverse tests to catch border cases and a wide spectrum of possible scenarios.

Integration testing is a testing method with which interfaces between components are checked. Integration testing takes modules that have been unit-tested and groups them together in larger aggregates and then tests those aggregates. Modules can be tested in an iterative fashion (one is inserted after another into an aggregate), or alternatively all at once. The iterative approach is usually better because it makes it much easier to localize a problem as opposed to having one, big aggregate and no hint on where the potential problem arose. Integration testing delivers the integrated system, which is, after all tests

succeeded, ready for system testing.

System testing works to expose defects and issues in a complete system and tries to verify that it meets its requirements. This means that system testing of software or hardware is applied to a complete, integrated system. Its main goal is to evaluate the system's compliance with its specified requirements. No knowledge of the inner structures and design implementation is required to perform system testing, which is why it falls within the scope of black-box testing. System testing seeks to detect defects both between the interfaces between aggregates and also within the system as a whole.

Additionally, regression testing has a focus on finding mistakes in the source code or behavior of the program after a major code change has been performed. This is to say, it seeks to uncover software regressions or old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly stops working as intended. Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Common methods of regression testing include re-running previously performed tests and checking whether previously fixed faults have reemerged. The depth of testing depends on many circumstances including but not limited to the phase in the development process, and the risk of already fixed bugs coming back to life again.

The testing done during this thesis is mainly in the form of integration and system testing. As will be explained in Section 2.2, tests are designed that attempt to verify the proper behavior of a given simulator back-end (hardware or software), which can both be interpreted as integration and system testing depending on the completeness of the system tested.

2. Comprehensive Test Framework

The test framework built while writing this thesis consists of two main parts: a general purpose test framework designed for universal incorporation and management of a variety of tests, and a set of high-level, PyNN-based, back-end-independent network tests.

The test framework serves the purpose of having a unified test management system, which is capable of simplifying and consistently controlling the overall testing procedure. The high-level network tests serve the purpose of actually investigating the functionality of the underlying back-end, that is, the hardware or software simulator.

2.1. The Test Framework Inspector

All code modules written for the testing and quality assurance of software that operates the FACETS hardware and its virtual versions are intended to be incorporated into one main test framework (see Figure 2.1), aptly called *Inspector*[1]. Inspector serves as a central test management system, which facilitates adding, removing, and running tests with a simple command line call. The framework can be utilized for unit-, integration-, system-, and regression-testing.

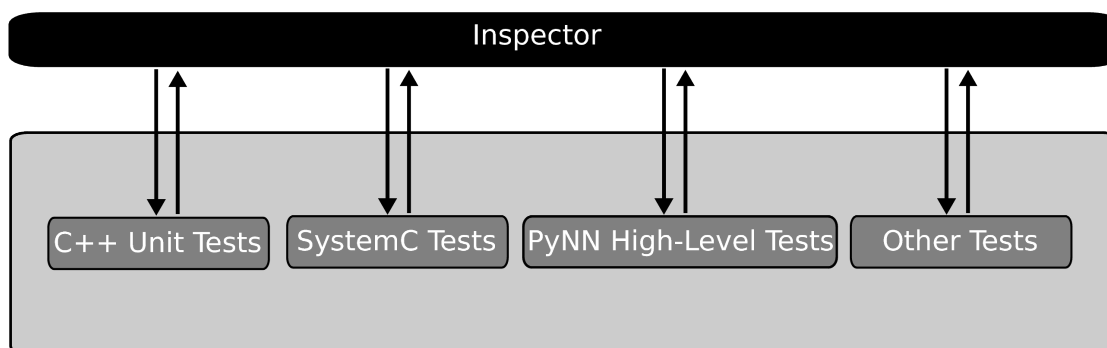

Figure 2.1.: A schematic view of the Inspector framework.

[1]This name has been chosen instead of simply referring to it as test framework or tests to distinguish it from individual tests and to allow a clear distinction between the framework itself and the underlying tests.

2.1.1. Requirements for the Framework

The framework must fulfill the following requirements:

- Robustness: the framework must behave correctly under unexpected circumstances, which are often provoked by software tests. It must handle misbehaving or crashing tests, permission errors, and similar situations in a graceful manner.
- User-friendliness: the framework must be easy to use. A user-unfriendly system demotivates users and most will refrain from using it in the future.
- Uncomplicated maintenance: adding, removing, or modifying tests must be easy and possible without heavy modifications.
- Universality: tests written in completely different programming languages covering completely different aspects should be easily installable into the framework.

2.1.2. Choice of Language

When it comes to the programming language that is to be used for the framework, there are two prime candidates: *C++ [53]* and *Python* (for the latter see Section 1.2.4). They are good candidates because they are already widely used within the global programming community and represent the languages mainly used throughout the existing software modules that operate the FACETS hardware. Both have their own advantages and disadvantages, which makes them both valuable in different programming scenarios. C++ is a hardware-oriented programming language. Many consider it to be not as high-level as Python for this very reason. Additionally, Python offers manifold built-in functionality and third-party packages [2, 32, 33, 38, 28, 54], which is not part of the C++ standard library. Python is, however, usually not as fast as C++, as it is in most cases interpreted byte-code.[2]

All these advantages and disadvantages must be taken into consideration together with the requirements outlined before.

Considering the robustness requirement, one can argue that there are no great differences between the two languages. Both offer debugging facilities and as long as good coding practices are followed, no language appears to be significantly more robust. However, one may also argue that Python is more high-level (e.g. it does not deal with pointers as C++ does) and this by itself may reduce the risk of common errors such as buffer or integer overflows. Also, high-level languages tend to be more easily readable and understandable

[2] There are some projects like Psyco or Unladen Swallow from Google which provide just-in-time-compilation for Python code, making it reportedly nearly as fast as compiled C or C++ code.

when the source code is read by someone who has not worked on the code before. All this can decrease the likelihood of mistakes being made by someone new to the project.

User-friendliness is a result of a good user interface and since this is provided by operating system functions, it does not really depend on the programming language used, but on how the programmer uses the operating system functions to construct a working user interface.

As for the requirement of uncomplicated maintenance, one may argue that Python has an advantage, because it does not require recompilation of the script once something has changed.

Taking all of this into consideration, Python has been chosen for the test framework because of the aforementioned advantages and due to its heavy use within the Electronic Vision(s) group. The latter factor enables other group members to easily understand and maintain the framework.

2.1.3. The Interface between the Framework and Tests

The next step is to develop a specification of the interface between the framework and individual tests. This interface must fulfill the requirements outlined before (see Section 2.1.1).

All tests must transfer information about their outcome to the main test framework. There are many ways to achieve this: files, pipes, network protocols, POSIX[3] [26] signals, and/or exit codes. Each method has advantages and downsides.

Files allow for arbitrary data to be exchanged, are extremely flexible, and in theory unlimited in their size. However, problems can arise as they need to be created and parsed correctly and are subject to race conditions in certain situations.

Pipes have similar advantages and disadvantages as files, as do network protocols, which need to be specified exactly and thoroughly in order to guarantee a smooth interaction between clients.

POSIX signals are asynchronous and simple, yet they can be difficult to handle in programming languages which are not low-level.

Exit-codes are, however, universal across all POSIX operating systems (which includes Windows, Unix, and Unix-like operating systems like Linux), are relatively simple and can be used as an indicator on whether a process finished successfully or not. This is why exit codes have been chosen as a way for the tests to communicate their results to the main test framework.

[3] Portable Operating System Interface [for Unix]

2.1.4. The Design of the Framework

The framework has been written in Python, and is using exit codes as a central means of communicating test results. Individual tests are executed from within the framework and each test returns with a proper exit code indicating the success or failure of the test. This approach has been motivated in the previous sections (see Sections 2.1.1, 2.1.2, and 2.1.3) and is considered suitable for this type of application.

The framework is embedded in a single, powerful, and platform-independent Python script. Its platform-independence has been given great priority, because it may be necessary to run the framework on different platforms, which may not just be limited to Unix or Unix-like systems. Having to redesign and/or rewrite the framework for every new platform it should run on is not desirable as it would cause additional development costs.

In order to achieve platform independence one has to exercise caution regarding what available features are being used. For example, even simple tasks as coloring output depends on the operating system. On Linux, escape codes can be used, which most likely do not work on Windows. Windows, though, offers several API calls to change the formatting of the output, which are not available on Linux.

The decision has been made to allow Unix-specific features, but the script can work perfectly without them and they can be disabled by the command line flag *–no-unix*.

The Python script must be executable, that is, its corresponding operating system- and file-system-specific executable flag(s) must be set appropriately. It can then be run from the command line without having to directly invoke the Python interpreter, as the interpreter is automatically called by the shell when it looks at the first line of the script, the so-called *she-bang line*. It specifies that the script is a Python script by pointing to a Python interpreter in the file system.

There also needs to be a configurable data structure, which can be manipulated by developers in order to add and remove tests, so that the framework knows about them. There are several ways to do that, including but not limited to using a configuration file or using hard-coded variables from within the Python script.

While the approach of using a configuration file appears to be more clean, there are disadvantages to this approach. One would need to write a parser for the configuration file. This parser needs to check for correct syntax and build a data structure out of the configuration file, which represents the data in an abstract format in memory. The parser needs to be robust and handle unexpected circumstances gracefully. Even when using third party libraries to simplify this development process, mistakes are still hard to avoid.

A better approach in this scenario is to directly embed the configuration and available

tests into the Python script itself in form of Python variables. This almost completely eradicates the process of having to correctly parse a configuration file, since the parsing of Python variables is done by the Python interpreter.

Within the framework script, a test variable in the global namespace holds all the available tests. It is called `tests`:

```
tests = (
("hierarchy.sub-hierarchy.test-1", "command_1"),
("group.subgroup.test-4", "another_command parameter"),
("group.another-subgroup.test-6", "make"))
```

As one can see from the example above, the global test variable is a tuple of tuples of two strings. The first string is a *test identifier*, the second string contains the *command* to be executed.

The test identifier can basically be anything, except that it must not start with a hyphen or it is confused for a command line option. Test identifiers can be thought of as names, giving each test a descriptive name that makes it unnecessary to remember and repeatedly type in the underlying command, the same way a domain name makes it unnecessary to remember the IP address[4] it resolves to. Test identifiers can themselves contain a hierarchy, where the hierarchy is conveyed by a delimiter in form of a character. While this character is a dot in the above example, any character can be chosen as the framework itself does not have any notion of a hierarchy or a delimiter for that matter and treats them like any other character. It merely serves as a visual aid to the developer.

The command to be executed, contained in the second string, is sent to the shell. These commands should invoke executable files. Shell/environment variables are expanded when the command is evaluated according to shell rules, so these variables should be directly usable in commands. The return value of the command will determine if the test succeeded. If 0 is returned the test succeeded, otherwise it failed. Additionally, the test should output relevant data to standard out/error that can later be examined in case a test failed.

2.2. High-Level Network Tests

Several neuronal network tests have been conceived, whose development and reasoning are described in the following sections.

[4] Internet Protocol address

2.2.1. Requirements

In order for tests to cover a whole array of back-ends, it is necessary to write them in a description language which allows for easy substitution of back-ends without heavy modifications to the testing code. For this reason, the use of PyNN has been motivated in previous sections (see Section 1.2.4). It is therefore a sound decision to use PyNN as the description language for this thesis. PyNN's low-level API is used exclusively throughout the tests to avoid interference with the test cases by another level of abstraction.

Additionally, these high-level tests should cover scenarios, which low-level unit tests cannot cover. This means that high-level tests should attempt to verify the correct implementation of the complex behavior of neuronal networks, which is the result of the dynamic interplay of elementary components already covered by low-level tests.

The tests should also be as efficient and encompass as much back-end configurations as possible. The main goal is to cover the complete set of circuits involved in the synaptic transmission path and back-end neurons. At the same time hardware fluctuations must be taken into account, so differential behavior must be evaluated instead of only absolute values.

2.2.2. Constructing a Versatile Test Architecture

Section 1.1.2 already introduced the reader to the idea of neuron models. Noteworthy is the fact that each neuron and synapse has a set of parameters which ultimately determines how the neuron or synapse responds to input. This set of parameters may be limited by the underlying back-end, but biologically realistic value ranges are supported by all back-ends tested for this thesis. As a prime example for such a parameter the threshold voltage can be named. It specifies the membrane potential necessary for the neuron to trigger an action potential and send it along its axon (see Section 1.1.1 and 1.1.2).

Changing the parameters of a neuron changes also its likelihood of releasing an action potential (with some exceptions and keeping in mind the refractory period described in Section 1.1). Sticking to the threshold voltage as the example parameter, it can be observed that increasing the threshold value causes less action potentials to be produced given the same amount of input to the neuron. And it is not hard to see why: increasing the threshold voltage also increases the voltage difference between the resting potential and the threshold potential. And the greater the difference the more EPSPs need to arrive at the neuron in a given amount of time in order for an outgoing action potential to occur. Here one can already see that manipulating the parameter set, even that of a single neuron, should manifest itself in a different firing rate.

Of course, the threshold potential is not the only parameter which can be modified. It is also possible to vary the weight of the synapse which delivers PSPs to the neuron in question. Changing the weight to a higher value than before also ought to increase the firing rate. A greater weight will lead to PSPs greater in amplitude reaching the neuron and hence increases the probability of an outgoing action potential to be fired.

The rate of incoming PSPs per synapse is also a parameter which lends itself to be shifted. In general it holds true that the higher the rate of incoming EPSPs the greater the rate of outgoing action potentials. This, however, is not always the case. For example if the neuron is already saturated by input and fires at its maximum rate, which is mostly determined by its absolute refractory period, the rate of outgoing action potentials does not change significantly.

Not only the rate of incoming EPSPs per synapse but also the number of synapses feeding the neuron can be changed. The higher the number of excitatory synapses, the higher the rate of outgoing action potentials, and the higher the number of inhibitory synapses, the lower the rate of outgoing action potentials, all factors being equal and keeping in mind the already discussed (see Section 1.1.1 and 1.1.2) limitations brought forth by the absolute refractory period.

All mentioned parameter changes should reveal themselves by an increased or decreased action potential rate–independent of the back-end. This can be verified by a relatively simple neuronal network consisting of just a single neuron fed by one or more external stimuli which are connected to the neuron with excitatory and/or inhibitory synapses and generate PSPs randomly following a Poisson distribution. Figure 2.2 shows the schematic setup.

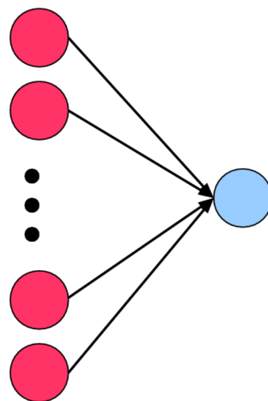

Figure 2.2.: Several stimuli connected to a single neuron.

For the tests to be more efficient, the neuron parameter set is generated randomly and is usually different each time the test is run. This allows a whole range of parameters to be swept, hence highly increasing the efficiency of the tests.

To this point only synapses connecting stimuli to neurons are tested, but inter-neuronal synapses are neglected. To avoid this the test setup can be easily extended to incorporate a second neuron, which uses an inter-neuronal synapse to connect to the first neuron. Now, both neurons can be checked for the expected change in action potential rates and both inter-neuronal and stimuli-to-neuron synapses are utilized. Figure 2.3 shows the setup.

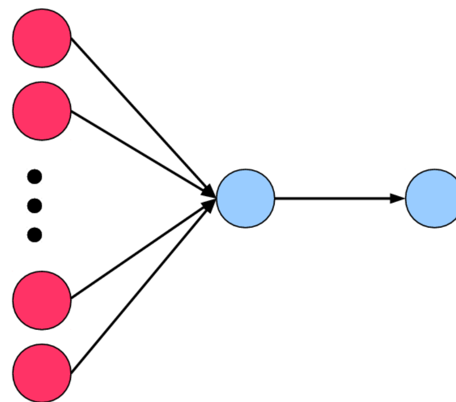

Figure 2.3.: Several stimuli connected to a neuron, which itself is connected to another neuron.

The random number generators applied within some mapping algorithms (see Section 1.2.4), like that designed for the Stage 2 hardware, which map the biological neuron to a hardware neuron, can be initialized with a new seed for every test. This will randomize the mapping but will, however, not prevent the mapping algorithm from choosing the very same location for a neuron or a location in close vicinity for every new run. This shortcoming would result in the tests losing efficiency, because the area tested on the real hardware is highly limited.

This problem can be overcome by creating a sufficiently large number of unused neurons, which serve the purpose of forcing the mapping algorithm to evenly distribute the complete set of neurons over the whole available space. Such a technique, together with using a new random-number-generator seed for every new test instance, should lead to a different location for all neurons every run. Ramifications of parameter changes will thus be tested against expected behavior in different locations of the hardware and, together with the intended randomness of parameter values, running such a test an infinite number of times

will test all available neuron places for correct behavior and the complete parameter ranges.

The unused neurons are, however, wasted and some of which may be much better utilized by using them as additional pairs. This would allow for running multiple test pairs in parallel. This leads to the final setup that is actually being used and sketched in schematic manner in Figure 2.4.

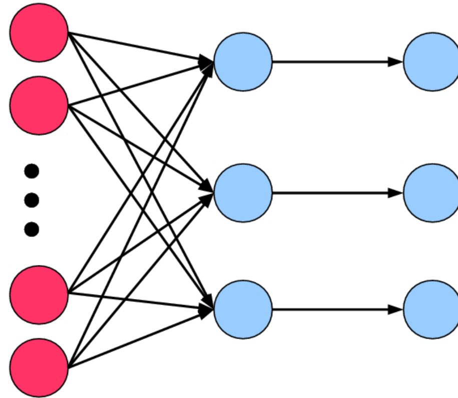

Figure 2.4.: Several pairs stimulated by the same source

The general test runs as follows. The user or an automated system like the test framework Inspector described in Section 2.1 starts the test specifying additional parameters such as the simulation time, the desired α factor (see Section 2.2.3 for further explanation), the back-end, number of excitatory and inhibitory stimuli and with what probability they should be connected to the neuron pairs, the number of pairs and unused neurons, and which parameter should be modified (see Section 2.2.4). After starting the script, the random number generator is seeded with the current time or a user-specified seed for reproducibility purposes. Following that a neuron and stimuli parameter set is chosen at random within biologically sound limits. The set of pair and unused neurons is created at once, so is the set of excitatory stimuli, followed by the set of inhibitory stimuli. The neurons used for the pairs are chosen randomly and connected appropriately to both stimuli and the other neuron of a given pair. The mapping algorithm should now have correctly mapped the PyNN description of the neuronal network to the correct back-end configuration. This is tested by starting the simulation of the neuronal network and recording the action potential rates of both neurons of each pair. After the simulation, the exact same steps are repeated with the exact same random-number-generator seed, but now one neuron, stimuli, or synapse

parameter is changed, so that the change should result in less action potential rates. This is verified in the second run. For a list of available parameter changes see Section 2.2.4. Section 2.2.3 goes into more detail on how it can be determined whether a change in action potential rates is statistically significant.

2.2.3. Statistical Considerations

In order to see if changes of neuronal parameters have any effect, two runs need to be done. The first run uses a given set of parameters, while the second run uses the same set except for a single parameter which is slightly altered (for a list of available changes see Section 2.2.4). The number of action potentials generated by neuron pairs is recorded and evaluated. Each change in neuronal parameter is designed so that the second run is expected to yield less action potentials. It is, however, not sufficient to simply compare the rate of action potentials triggered and declare the test as successful if the rate decreased in the second run. It must also be determined whether the change is statistically significant.

In the following, the simulation time is always assumed to be the same. For a discrete number of action potentials, denoted by n_i, and the assumption that the firing is a Poisson process, the error of this expectation value estimator is given by

$$\Delta n_i = \sqrt{n_i} \tag{2.1}$$

For the difference $n_{diff-i,j}$ between two action potential counts n_i and n_j Gaussian error propagation dictates

$$\Delta n_{diff-i,j} = \sqrt{(\Delta n_i)^2 + (\Delta n_j)^2} \tag{2.2}$$

which can be simplified to

$$\Delta n_{diff-i,j} = \sqrt{n_i + n_j} \tag{2.3}$$

Taking this into account, one can make assumptions about the likelihood that a given difference in action potential activity between the first and second run is pure coincidence or the result of the correct realization of a parameter change. Because the number of action potentials is assumed to be large, the distribution can be approximated by a Gaussian. The area under the Gaussian bell curve is given by the error function erf.

Thus, given a difference in action potential count $n_{diff-i,j}$ and its error $\Delta n_{diff-i,j}$, and the introduction of a factor α which is defined by

$$n_{diff-i,j} = \alpha \cdot \Delta n_{diff-i,j} \tag{2.4}$$

one can calculate the probability of this change being significant by

$$p = erf(\frac{\alpha}{\sqrt{2}}) \tag{2.5}$$

Equation 2.5 is the basis for Table 2.2 which shows values for α factors between 1 and 6, inclusive.

α	$erf(\frac{\alpha}{\sqrt{2}})$
1	0.682689492137
2	0.954499736104
3	0.997300203937
4	0.999936657516
5	0.999999426697
6	0.999999998027

Table 2.2.: Error function values to 12 decimal places

Hence, a α factor of 3 means that the change in action potential count is with an approximate probability of 99.73% significant.

One can require a certain minimum α for the tests to be declared as successful. This can actually be done on the command line.

2.2.4. The Test Cases: Varying Constituents and Connections

This section lists all available parameter variations. Additional parameter variations are possible but not implemented as of the time this has been written. Each step in the mapping of a description of a neuronal network architecture to an actual hardware configuration can be erroneous (see Section 1.2.4). The goal of the parameter variations presented in the following sections is to check whether neurons and synapses are mapped at all, if the corresponding mapping tables hold consistent index pairs, and if so, whether the individual parameter values that belong to the units are mapped in a qualitatively correct way.

Increase in Threshold Potential

The threshold potential is a critical value of the membrane potential to which a neuron must be depolarized to initiate an action potential (see Sections 1.1.1 and 1.1.2). All

other things being equal, the higher the threshold potential the less outgoing spikes are generated by the neuron as a result of the stimulation (keeping in mind the refractory period as described in Section 1.1.1).

The test is run for a given configuration, then the same test is run with a higher threshold potential. Each time the outgoing spike activity is recorded and evaluated. If the number of spikes decreases significantly (by means of the α measure introduced in Section 2.2.3) once a higher threshold potential is set, the test counts as being successful, otherwise it failed.

Increase in Stimuli Rate

All other things being equal, the higher the stimuli frequency the more outgoing spikes should be generated by the neuron as a result of the stimulation (again keeping in mind the refractory period as described in Section 1.1.1). With this supported parameter variation, the test is run for a given configuration, then the same test is run with a lower stimulation frequency. Each time the outgoing spike activity is recorded and evaluated. If the number of outgoing spikes decreases significantly when a lower stimuli rate is set, the test counts as being successful, otherwise it failed.

Switching of Synapse Type from Excitatory to Inhibitory

The mapping of the correct type of synaptic link is also important (see Sections 1.1.1 and 1.1.2). The erroneous mapping of excitatory synapses in the description of the neuronal network to inhibitory synapses on the back-end is covered with this test. If excitatory synapses are used between stimuli and neuron, and between the neurons of each pair, outgoing spikes should be produced. If the synapse type between two neurons of a neuron pair is switched to inhibitory, spike activity is expected to cease. If this is the case, the test succeeded, otherwise it failed.

Decrease in Synapse Weight

The weight of synapses must also be correctly mapped to the back-end. Here, too, mistakes can happen in the form of hardware failures or incorrect configurations. This supported parameter change attempts to cover this possibility. If the synaptic weight is decreased, the spike activity of both neurons of a pair should decrease significantly. If this is the case, the test succeeded, otherwise it failed.

Decrease in Number of Excitatory Synapses

Decreasing the number of excitatory synapses should result in less spikes being produced by both neurons of a pair. This test attempts to verify this behavior. The purpose, again, is to detect if synapses are mapped correctly. If they are, removing one of them should lead to a different spike frequency. If the synapse that is removed did not exist on the hardware in the first place, for example due to an erroneous mapping procedure, no change in outgoing spike activity should be recorded and the test fails.

2.2.5. Integration of Tests into the Test Framework

The above tests are incorporated into the main test framework. This allows for efficient execution of single tests or the whole set of tests either explicitly or implicitly when changing relevant code. The tests are included in the `inspector.py` script file.

The test identifiers contain the back-end the tests will run on, e.g. `stage1` for the FACETS Stage 1 hardware, `stage2ess` for the Executable System Specification of the FACETS Stage 2 hardware.

The tests have been successfully applied for this thesis. The functionality has been verified via software simulators, as is shown in Section 3.1.

3. Application of Tests

The purpose of this chapter is to show that the developed high-level tests (see Section 2.2) can indeed run on multiple simulator back-ends and are useful for finding mapping, configuration or back-end problems.

3.1. Testing NEST and NEURON

The software back-ends NEST and NEURON (see Section 1.2.3) have already been well-tested by the neuroscience community. This makes them an able tool to verify the validity of the developed high-level neuronal network tests.

First, a demonstrator sweep is performed to qualitatively verify the behavior of the tests. Table 3.2 shows the used parameters.

Parameter	Value
Variation	Threshold Potential
Simulation time	700 ms
Pairs	4
Unused neurons	10
Maximum number of excitatory stimuli	100
Maximum number of inhibitory stimuli	10
Stimuli connection probability	0.3

Table 3.2.: Parameters chosen for a demonstrator run

While these parameters seem to be random at first glance, they are motivated by the fact that they turned out to be good for showing the qualitative behavior of the neuronal network in a way that is even visible to the naked eye when studying the corresponding spike output in a raster plot. Figure 3.1 shows such plots resulting from running this scenario with NEST.

The figure shows the output spike times of the second cell over the full duration of the experiment. The same parameter set is used in both runs, except for an increase in the threshold potential from -55.62 mV to -50.41 mV in the second run. This is expected

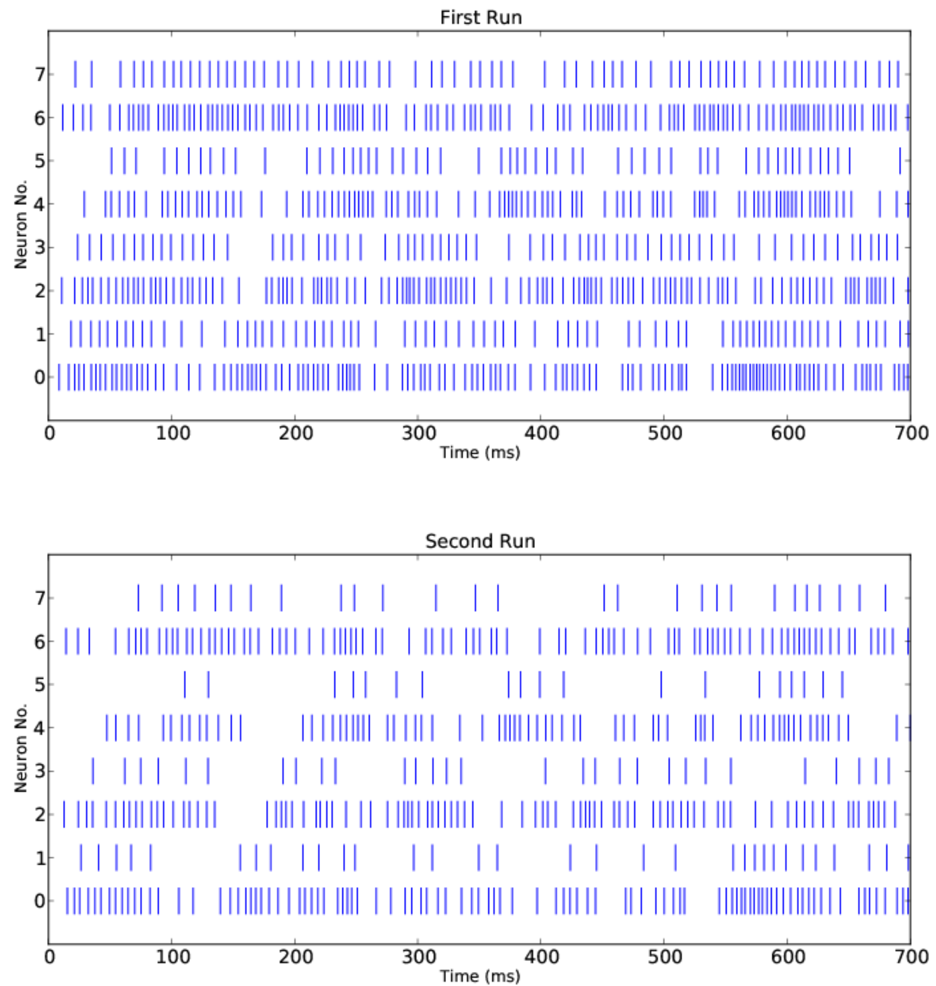

Figure 3.1.: Effect of changing the threshold potential on NEST from the first to the second run. The abscissa shows the simulation time and the ordinate the neuron number. Each even neuron number is the neuron that is directly fed by the stimuli and each odd neuron number is the one that is only connected to the stimuli-excited neuron. Neuron 0 and neuron 1 make up a pair, so do neuron 2 and neuron 3, and so on.

to reduce the resulting output spike frequency, which is indeed clearly visible in the shown plots. It is also confirmed by an alpha factor of above 2 for every neuron (see Section 2.2.3), which is a remarkably high significance for such a short simulation time.

Now that the general, qualitative effect of the tests has been demonstrated, a more biologically realistic scenario is chosen. Table 3.4 and 3.6 show the used parameters.

Parameter	Value
Variation	Threshold Potential, Stimuli Rate, Synaptic Weight, Synapse Type
Simulation time	10000 ms
Pairs	10
Unused neurons	10
Maximum number of excitatory stimuli	800
Maximum number of inhibitory stimuli	200
Stimuli connection probability	0.3

Table 3.4.: Parameters chosen for variations of the threshhold potential, stimuli firing rate, synaptic weight, and synapse type.

Parameter	Value
Variation	Number of Excitatory Stimuli
Simulation time	100000 ms
Pairs	10
Unused neurons	10
Maximum number of excitatory stimuli	8
Maximum number of inhibitory stimuli	2
Stimuli connection probability	1.0

Table 3.6.: Parameters chosen for variation of the number of excitatory stimuli.

The number of stimuli is reasonable as it provides a good compromise between biological realism and performance considerations; only when the number of excitatory stimuli is varied, a much lower stimuli number is chosen in order for the tests to succeed in less time. The number of stimuli together with the usual value range of neuron parameters leads to an approximate rate of produced spikes. In order to measure a significant difference at this rate, a simulation time of 10000 ms (100000 ms for the stimuli number variation) is usually a good setting. Then the various neuronal and synaptic parameter variations are

performed on both NEST and NEURON as outlined in Section 2.2.4.

The average number of action potentials produced by the second neuron of a pair and the resultant average α factor are shown in Table 3.8. As can be seen from the table, all α factors are well above 12 meaning that the chance of any of these changes being pure coincidence instead of being ramifications of the parameter variations is well below $3.55 \cdot 10^{-33}$. The tests have hence demonstrated their validity.

Variation	Average spike count in first run	Average spike count in second run	Average Alpha
Threshold Potential	6150	4818	12.71
Stimuli Rate	6642	4731	17.92
Synaptic Weight	4879	3759	12.05
Synapse Type	6608	0	81.29
Number of Exc. Stimuli	11033	9287	12.25

Table 3.8.: Effects of parameter variations on the second neuron of a pair in NEST.

The same test configurations have also been run on the NEURON simulator (see Section 1.2.3) with qualitatively identical results, but for clarity and readability purposes only data generated with NEST is shown here.

3.2. Testing the FACETS Stage 1 Hardware

The Spikey chip introduced in Section 1.2.1 has also been tested and the results are shown in this section.

First, a demonstrator run has been performed to again check the qualitative behavior. Table 3.10 shows the used parameters.

Parameter	Value
Variation	Threshold Potential, Stimuli Rate, Synaptic Weight, Synapse Type, Number of Exc. Stimuli
Simulation time	500 ms
Pairs	4
Unused neurons	10
Maximum number of excitatory stimuli	90
Maximum number of inhibitory stimuli	10
Stimuli connection probability	0.3

Table 3.10.: Parameters chosen for the Stage 1 demonstrator run.

These values are motivated by the fact that they are good for producing easily analyzable plots on the Spikey chip, as can be seen in Figure 3.2.

It plots the output spike trains generated during the two runs. Again, the same parameter set is used in both runs, except for an increase in the threshold potential from -63.98 mV to -58.68 mV in the second run. This is expected to decrease the resulting output spike frequency, which is indeed clearly visible in the plots. It is also confirmed by the difference in spike activity yielding an α factor of above 4 for every neuron (see Section 2.2.3).

After demonstrating the qualitative effect of the tests on the Stage 1 hardware, two Spikey chips are tested more thoroughly. Spikey version 4 offers 192 available neurons, leading to 96 available pairs. Since every hardware neuron has a chance to be representing the second cell in such a pair, it possibly has to be fed back into the chip and thereby occupy a synapse driver. Thus, 64 synapse drivers remain to be used for the external stimuli.

Using this many pairs of neurons firing at relevant output rates will, however, make the tests fail. This is due to the fact that the number of produced spikes can be too large for the chip-to-FPGA communication protocol to handle [8, Section 4.3.7]. Hence, a lower number of pairs must be used and the test must be run multiple times to cover the whole chip. All available parameter variations are tested on both chips.

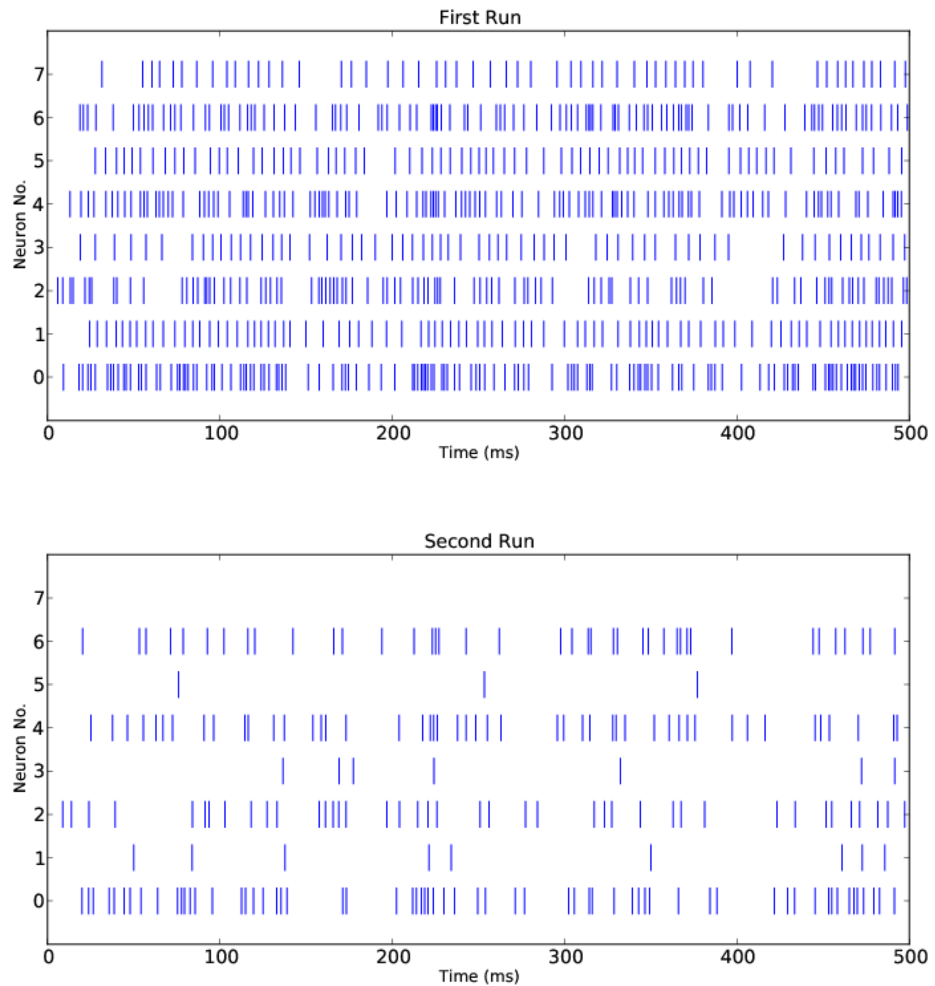

Figure 3.2.: Effect of changing the threshold potential on Stage 1 from the first to the second run. The abscissa shows the simulation time and the ordinate the neuron number. Each even neuron number is the neuron that is directly fed by the stimuli and each odd neuron number is the one that is only connected to the stimuli-excited neuron. Neuron 0 and neuron 1 make up a pair, so do neuron 2 and neuron 3, and so on.

Successful Fault Identification

First the chip with the individual identification number 415 is tested. Table 3.12 shows an example of produced spikes by the second neuron when varying the threshold potential in the second run. It is shown that the number of spikes decreases in the second run as is intended. All tests succeed on this chip.

Neuron Index	Total spike count in first run	Total spike count in second run	Alpha
0	1953	1387	9.79
1	1125	310	21.51
2	2460	1735	11.19
3	365	13	18.10
4	2514	1722	12.17
5	289	8	16.31

Table 3.12.: Total spikes produced by some neurons on the Spikey chip 415.

The same test run is performed on the chip with the identification number 456. Some neurons on this chip, however, do not produce any spikes at all, as shown in Table 3.14. Repeatedly running the same test reveals that the neurons that do not spike are always the same.

Neuron Index	Total spike count in first run	Total spike count in second run	Alpha
0	371	289	3.19
1	107	62	3.46
2	0	0	-
3	0	0	-
4	346	271	3.02
5	161	111	3.03

Table 3.14.: Total spikes produced by some neurons on the Spikey chip 456.

The run of the high-level neuronal network tests have shown two, probably unavoidable problems with the Spikey chip. First, the number of spikes the chip is able to record does not allow a complete utilization of the chip if neurons produce too many spikes. This limit is imposed by the hardware and cannot be avoided in general without a system redesign. The second problem is the apparent degradation and failure of neurons, which is an inevitable

consequence of the usage of electronic circuits. In this situation, the presented tests can serve as a tool to identify the faulty circuits and exclude them in following mapping runs.

3.3. Testing with the FACETS Stage 2 Executable System Specification

As a positive side effect of dealing with the Executable System Specification of the FACETS Stage 2 project (see Section 1.2.2), the overall structure of the Executable System Specification source code has been improved in collaboration with Bernhard Vogginger as part of this thesis. These numerous changes include but are not limited to: increasing readability to facilitate finding problems, adjusting visibility of class members to better reflect good coding practices and to conform with the object-oriented programming philosophy of encapsulation, optimizing code, erasing unnecessary or redundant statements, and commenting the resultant code.

After these changes have been made, the high-level neuronal network tests (see Section 2.2) have been run on the Stage 2 virtual hardware. A demonstrator plot is shown in Figure 3.3. Table 3.16 shows the used parameters.

Parameter	Value
Variation	Stimulus Rate
Simulation time	2000 ms
Pairs	4
Unused neurons	80
Maximum number of excitatory stimuli	20
Maximum number of inhibitory stimuli	8
Stimuli connection probability	0.5

Table 3.16.: Parameters chosen for the Stage 2 Executable System Specification demonstrator run.

Here it can be seen that the tests also work on the Stage 2 virtual hardware.

After a demonstrator run, the tests have been put to use and helped in finding problems in the software stack that configures the virtual hardware and will also configure the real hardware in the future.

The tests revealed that the Stage 2 system, too, is limited in the number of recordable spikes. If too many pairs are created and there are only few unused neurons, the pair neurons are not distributed widely across the wafer and too many neurons send on the

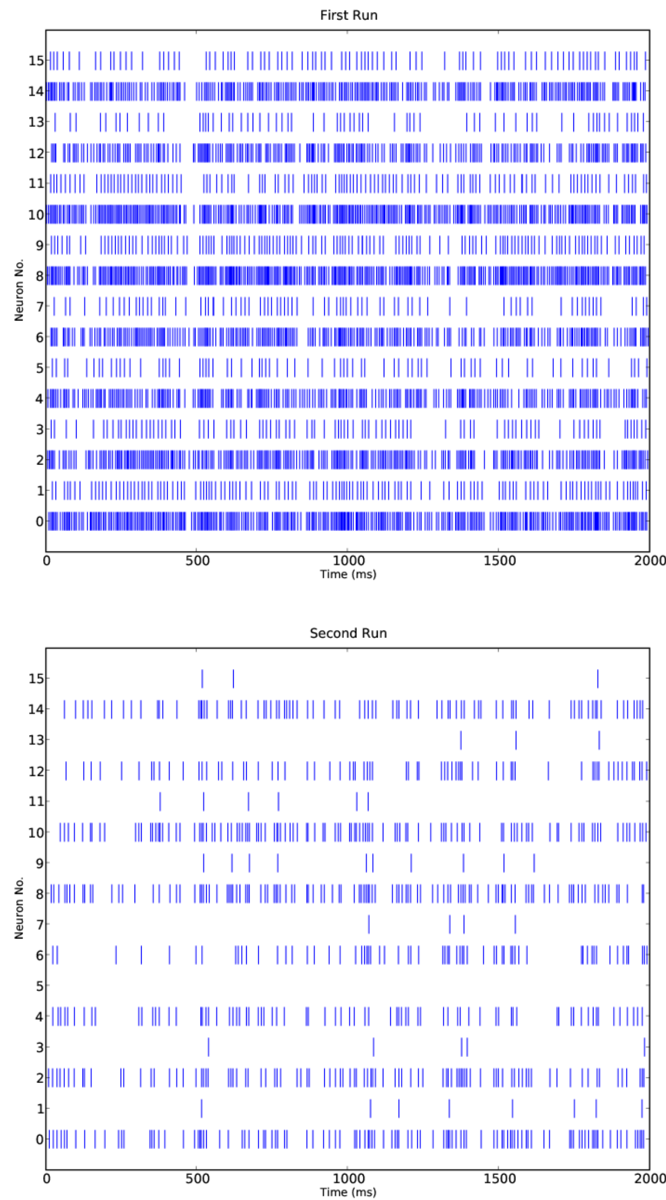

Figure 3.3.: Effect of changing the stimuli rate on the Stage 2 virtual hardware from the first to second run. The abscissa shows the simulation time and the ordinate the neuron number. Each even neuron number is the neuron that is directly fed by the stimuli and each odd neuron number is the one that is only connected to the stimuli-excited neuron. Neuron 0 and neuron 1 make up a pair, so do neuron 2 and neuron 3, and so on.

same Layer 2 connection. This leads to congestion of the Layer 2 connection and results in a loss of recorded spikes.

Successful Debugging

During the time this thesis has been written two additional problems have been found in collaboration with Bernhard Vogginger within the software stack operating the Executable System Specification. The software stack includes but is not limited to the PyNN API implementation, the mapping process, and the Executable System Specification itself (see Sections 1.2.4 and 1.2.2).

The first problem arose from a defective implementation of the *parameter transformation* (see Section 1.2.4) if a neuron has no outgoing connections to other neurons. This may happen, for example, when using the high-level PyNN API (see Section 1.2.4) to create two neuron populations and connect them with a certain probability below 1. Then, neurons may not have any outgoing connections.

In a HICANN (see Section 1.2.2) a synapse driver is designated to utilize all synapses it feeds in either exclusively inhibitory or exclusively excitatory fashion. Up to 64 neurons may send to a given driver. The parameter transformation determines whether a synapse driver has to be configured to be excitatory or inhibitory by checking the synapse type of the first outgoing synapse of the first neuron that is assigned to this driver.

The problem emerged in the defective implementation when the first neuron happened to have no outgoing connections at all. In this case the synapse driver was not configured at all, leading to a loss of all other synapses also using this driver. This problem had been demonstrated by the high-level neuronal network tests, when unused neurons had been created additionally to neuron pairs. The unused neurons can be configured to be not connected with each other and hence have no outgoing synaptic connections. This led to certain synapse drivers not being set up correctly and the tests failed.

This problem has been fixed by iterating through all neurons that use a given driver until one is found which has at least one outgoing connection. Fixing this problem prevents synapses from getting lost, which may be hard to track in more complex neuronal network architectures leading to unexpected results.

The second identified, in this case still unfixed problem surfaces during the *routing* of on-wafer connections (see Section 1.2.4) in case there are neurons with no incoming connections. This can happen under similar conditions as the first problem, especially when two neuron populations are connected with a probability below 1. If no incoming connections are present, the synapse configuration in the HICANN is distorted.

As an example one may consider two neurons placed adjacent to each other in the HICANN. The first neuron is not stimulated, but the second one is. The routing algorithm ignores synapse columns for neurons with no incoming connections and instead erroneously places the synapse column of the *second* neuron for the first neuron. This means that the second neuron will not get the input it should get.

This problem, too, has been found by the execution of the high-level neuronal network tests.

Conclusion and Outlook

The presented thesis originates from the core problem that as complexity rises, the ability to spot mistakes and flaws in the implementation of models, especially those of recurrent neuronal networks, diminishes. It approaches this problem by providing methods that ensure the validity, functionality, and a maximum degree of accuracy of neuronal network model implementations. Proof of the versatility, applicability, and benefits of the high-level neuronal network tests is presented.

The tests presented (see Section 2.2) and applied (see Chapter 3) showed that the paradigm of mistakes manifesting themselves as behavior that is contrary to well-defined expectations is usually a sound one. It has been shown that applying even minimalistic (in terms of the number of involved components) tests can facilitate the identification of problems in code modules and imperfections in hardware. It can be argued that the minimalistic approach should even be strongly preferred to more complex ones, because if a test is too complicated one might not know if unexpected behavior is the result of a failure in the simulator back-end or a flawed assumption or implementation in the tests themselves. The paradigm of minimalistic but thorough tests is therefore a benefit and advantage in pinpointing potential problems with the back-end and the operating software: It avoids having to struggle with unnecessary complexity imposed by overly intricate neuronal architectures.

The use of PyNN has been motivated (see Section 1.2.4) and has helped in interfacing different back-ends. Neuron pairs are found to be the most basic unit that allows to cover the testing of neuronal parameters as well as stimuli-to-neuron and neuron-to-neuron synaptic connections in an efficient manner.

The spectrum of scenarios currently covered by the presented tests is, of course, limited. This arises from two factors: First, the limited time available for this thesis constrained the set of code and test scripts that could be generated. Second, the tests developed are most effective when being run as often as possible. This is a direct consequence of the randomness introduced in order to cover a whole array of mapping scenarios. If each test used the exact same parameter set, the resultant hardware configuration would always be the same and the testing would lose a great deal of efficiency.

The approach taken can be applied to forthcoming hardware and software simulators,

as it is generic in nature and not dependent on the back-end itself. It takes relatively low development time while at the same time yielding high test efficiency.

The work at hand constitutes the first step towards a complete quality assurance system for the FACETS neuromorphic modeling software framework and is believed to support, motivate, and initiate contribution of further tests, especially unit tests [3], more high-level tests, and possibly even benchmark models that evaluate functionality at a very high level. Some of these already exist [17] and may conceivably be integrated as modules into the test framework.

All tests presented should be run regularly, especially after source code changes of the mapping tool or virtual hardware. Results of these tests should be documented. The logging functionality of the test framework Inspector (see Section 2.1) can be used for this. Improvements such as fixing of coding mistakes and other problems should become apparent from the logged results and may provide further motivation to develop additional tests.

A long-term objective is to offer a broad range of tests to the neuroscience community to help make neuromorphic modeling back-ends more robust and error-free. This will aid in the constant development of neuromorphic modeling platforms and thereby help in creating a functional tool that will gain further insight into the inner workings of our brain.

A. Resources and Supplements

A.1. Source Code of the Test Framework

The source code of the test framework Inspector can be found in a git repository:

`git@gitviz.kip.uni-heidelberg.de:symap2ic.git`

The corresponding script is located at:

`symap2ic/tests/inspector.py`

A.2. Source Code of the High-Level Neuronal Tests

The source code of the high-level neuronal network tests can be found in a git repository:

`git@gitviz.kip.uni-heidelberg.de:symap2ic.git`

The corresponding scripts are located at:

`symap2ic/components/pynnhw/test/high_level_network_tests`

For the management and tracking of software bugs Indefero is used and accessible at:

`https://gitviz.kip.uni-heidelberg.de/index.php/p/symap2ic/`

The bug reports filed of the problems found with the high-level neuronal network tests can be found among others in:

`https://gitviz.kip.uni-heidelberg.de/index.php/p/symap2ic/issues/`

A.3. Using the Test Framework

The inspector framework is contained in a file called `inspector.py`. A short help function is built-in and available using the command line parameter `--help`.

Simply running all tests available can be done with calling Inspector without any arguments. This can take a while depending on the number of available tests.

If one wants to limit the test set that is executed, one can specify one or more masks. Masks are strings that must be present anywhere in a test identifier for a test command to be executed. For example the test masks `system-sim`, `pynn`, and `build` run all tests which have the strings `system-sim`, `pynn`, or `build` in them.

In order to just list but not execute tests, a dry run option is available and is accessible by the `-d` option.

The default behavior is that all tests are run, even if single tests fail. If one wants the test flow to be stopped if a test failed, the `-s` option is available.

If one wants to suppress test output and only wishes to see a summary, the quiet mode enabled with `-q` is available.

A log file can be specified by using the `-l=logfile.txt` option, where `logfile.txt` can be substituted by the desired file name for the log file. If the file already exists, it will be overwritten. The log file still contains the complete output, even in quiet mode.

By default, the terminal output is colored. If one wishes to disable this, the `--no-colors` option must be used. It is to be noted that log files are never colored, because some text editors and viewers cannot handle escape codes correctly or simply do not support them because they are Unix-specific.

If the framework is not running on a Unix system, the `--no-unix` option should be used. This disables Unix-specific functionality like interpreting negative return values as signal codes or using escape codes to color terminal output.

Tests are directly added, removed, or modified in the `inspector.py` executable. The global tests variable contains tuples. Each tuple contains two strings. The first is the test identifier, the second the test command. See Section 2.1.4 for further information.

Bibliography

[1] L. Abbott. Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain Res. Bull.*, 50:303–304, 1999. 1.1.2

[2] D. Abrahams and R.W. Grosse-Kunstleve. Building hybrid systems with Boost.Python, 2003. URL `http://www.boostpro.com/writing/bpl.pdf`. 2.1.2

[3] Marvin Albert. Liquid computing mit neuromorpher hardware. Bachelor thesis (German), University of Heidelberg, 2010. 3.3

[4] C. Beaulieu and M. Colonnier. A laminar analysis of the number of round-asymmetrical and flat-symmetrical synapses on spines, dendritic trunks, and cell bodies in area 17 of the cat. *J Comp Neurol*, 231(2):180–9, Jan 1985. 1.1.1

[5] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-80938-9. 1.3.2

[6] R. Brette and W. Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J. Neurophysiol.*, 94:3637 – 3642, 2005. doi: NA. 1.1.1, 1.1.2

[7] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. Harris Jr, M. Zirpe, T. Natschlager, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. P. Davison, S. El Boustani, and A. Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies, 2006. URL `http://arxiv.org/abs/q-bio.NC/0611089`. (document)

[8] Daniel Brüderle. *Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System*. PhD thesis, 2009. (document), 1.2.1, 3.2

[9] Daniel Brüderle, Eric Müller, Andrew Davison, Eilif Muller, Johannes Schemmel, and Karlheinz Meier. Establishing a novel modeling tool: A python-based interface for a neuromorphic hardware system. *Front. Neuroinform.*, 3(17), 2009. (document), 1.2.4

[10] Daniel Brüderle, Johannes Bill, Bernhard Kaplan, Jens Kremkow, Karlheinz Meier, Eric Müller, and Johannes Schemmel. Simulator-like exploration of cortical network architectures with a mixed-signal vlsi system. In *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10)*, 2010. 1.2.1

[11] Ted Carnevale. Neuron simulation environment. *Scholarpedia*, 2(6):1378, 2007. 1.2.3

[12] William J. Dally and John W. Poulton. *Digital systems engineering*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-59292-5. (document)

[13] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.*, 2(11), 2008. 1.2.4, 1.2.4

[14] Markus Diesmann and Marc-Oliver Gewaltig. NEST: An environment for neural systems simulations. In Theo Plesser and Volker Macho, editors, *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, volume 58 of *GWDG-Bericht*, pages 43–70. Ges. für Wiss. Datenverarbeitung, Göttingen, 2002. 1.2.3

[15] Rodney Douglas, Henry Markram, and Kevan Martin. *The Synaptic Organization in the Brain*, chapter Neocortex, pages 499–558. Oxford University Press, 5 edition, 2004. ISBN 0-19-515955-1. 1.1.1, 1

[16] M. Ehrlich, C. Mayr, H. Eisenreich, S. Henker, A. Srowig, A. Grübl, J. Schemmel, and R. Schüffny. Wafer-scale VLSI implementations of pulse coupled neural networks. In *Proceedings of the International Conference on Sensors, Circuits and Instrumentation Systems (SSD-07)*, March 2007. 1.2.2

[17] M. Ehrlich, K. Wendt, L. Zühl, R. Schüffny, D. Brüderle, E. Müller, and B. Vogginger. A software framework for mapping neural networks to a wafer-scale neuromorphic hardware system. In *Proceedings of the Artificial Neural Networks and Intelligent Information Processing Conference (ANNIIP) 2010*, pages 43–52, 2010. (document), 1.2.4, 3.3

[18] Jochen M. Eppler, Moritz Helias, Eilif Muller, Markus Diesmann, and Marc-Oliver Gewaltig. PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.*, 2(12), 2008. 1.2.3

[19] J. Fieres, J. Schemmel, and K. Meier. Realizing biological spiking network models in a configurable wafer-scale hardware system. In *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*, 2008. 1.2.4

[20] Wulfram Gerstner and Werner Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002. 1.1.1

[21] Marc-Oliver Gewaltig and Markus Diesmann. NEST (NEural Simulation Tool). *Scholarpedia*, 2(4):1430, 2007. 1.2.3

[22] Dan Goodman and Romain Brette. Brian: a simulator for spiking neural networks in Python. *Front. Neuroinform.*, 2(5), 2008. 1.2.3

[23] Paul Hamill. *Unit test frameworks*. O'Reilly, 2004. ISBN 0596006896. 1.3.2

[24] Donald O. Hebb. *The Organization of Behaviour*. Wiley, New York, 1949. 1.1.1

[25] Michael L. Hines and Nicholas T. Carnevale. *The NEURON Book*. Cambridge University Press, Cambridge, UK, 2006. ISBN 978-0521843218. 1.2.3

[26] IEEE. Standard for information technology - portable operating system interface (POSIX). shell and utilities. Technical report, IEEE, 2004. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1309816`. 2.1.3

[27] Renaud Jolivet, Ryota Kobayashi, Alexander Rauch, Richard Naud, Shigeru Shinomoto, and Wulfram Gerstner. A benchmark test for a quantitative assessment of simple neuron models. *Journal of Neuroscience Methods*, 169(2):417 – 424, 2008. ISSN 0165-0270. Methods for Computational Neuroscience. 1.1.2

[28] Eric Jones, Travis Oliphant, and Pearu Peterson. SciPy: Open source scientific tools for Python, 2001. URL `http://www.scipy.org/`. 2.1.2

[29] Bernhard Kaplan, Daniel Brüderle, Johannes Schemmel, and Karlheinz Meier. High-conductance states on a neuromorphic hardware system. In *Proceedings of the 2009 International Joint Conference on Neural Networks (IJCNN)*, 2009. 1.2.1

[30] C. Koch and I. Segev, editors. *Methods in Neuronal Modeling: From Ions to Networks, 2nd Edition*. The MIT Press, 1998. 1.1.2

[31] Arvind Kumar, Sven Schrader, Ad Aertsen, and Stefan Rotter. The high-conductance state of cortical networks. *Neural Computation*, 20(1):1–43, Jan 2008. 1.1.1

[32] Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer, 3rd edition, February 2008. ISBN 3540739157. 2.1.2

[33] Mark Lutz. *Programming Python: Object-Oriented Scripting.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001. ISBN 0596000855. Foreword By-Guido Van Rossum. 1.2.4, 2.1.2

[34] H. Markram, J. Lübke, and B. Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic aps. *Science*, 275:213–215, 1997. 1.1.1

[35] H. Markram, Y. Wang, and M. Tsodyks. Differential signaling via the same axon of neocortical pyramidal neurons. *Proceedings of the National Academy of Sciences of the United States of America*, 95(9):5323–5328, April 1998. ISSN 0027-8424. URL `http://view.ncbi.nlm.nih.gov/pubmed/9560274`. 1.1.1

[36] Morrison, Abigail, Diesmann, Markus, Gerstner, and Wulfram. Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics*, 98(6): 459–478, June 2008. ISSN 0340-1200. doi: 10.1007/s00422-008-0233-1. 1.1.1

[37] Richard Naud, Nicolas Marcille, Claudia Clopath, and Wulfram Gerstner. Firing patterns in the adaptive exponential integrate-and-fire model. *Biological Cybernetics*, 99(4):335–347, Nov 2008. doi: 10.1007/s00422-008-0264-7. URL `http://dx.doi.org/10.1007/s00422-008-0264-7`. 1.1.2

[38] Travis E. Oliphant. Python for scientific computing. *IEEE Computing in Science and Engineering*, 9(3):10–20, 2007. 2.1.2

[39] Dejan Pecevski and Thomas Natschläger. PCSIM website. `http://sourceforge.net/projects/pcsim`, 2008. 1.2.3

[40] Dejan A. Pecevski, Thomas Natschläger, and Klaus N. Schuch. Pcsim: A parallel simulation environment for neural circuits fully integrated with python. *Front. Neuroinform.*, 3(11), 2009. 1.2.3

[41] PyNN. A Python package for simulator-independent specification of neuronal network models – website. `http://www.neuralensemble.org/PyNN`, 2008. 1.2.4

[42] Python. The Python Programming Language – website. `http://www.python.org`, 2009. 1.2.4

[43] S Ramon y Cajal. *Histologie du Systeme Nerveux de l'homme et des Vertebres.* 1911. 1.1.1

[44] Sylvie Renaud, Jean Tomas, Yannick Bornat, Adel Daouzli, and Sylvain Saighi. Neuromimetic ICs with analog cores: an alternative for simulating spiking neural networks. In *Proceedings of the 2007 IEEE Symposium on Circuits and Systems (ISCAS2007)*, 2007. (document)

[45] F. Rieke, D. Warland, R. de Ruyter van Steveninck, and W. Bialek. *Spikes - Exploring the neural code.* MIT Press, Cambridge, MA., 1997. 1.1.1

[46] Guido Van Rossum. *Python Reference Manual: February 19, 1999, Release 1.5.2.* iUniverse, Incorporated, 2000. ISBN 1583483748. 1.2.4

[47] J. Schemmel, A. Grübl, K. Meier, and E. Muller. Implementing synaptic plasticity in a VLSI spiking neural network model. In *Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN'06)*. IEEE Press, 2006. (document), 1.2.1

[48] J. Schemmel, D. Brüderle, K. Meier, and B. Ostendorf. Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS'07)*. IEEE Press, 2007. (document), 1.2.1

[49] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*, 2008. (document), 1.2.2

[50] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10)*, 2010. (document), 1.2.2

[51] Michael Shelley, David McLaughlin, Robert Shapley, and Jacob Wielaard. States of high conductance in a large-scale model of the visual cortex. *J. Comp. Neurosci.*, 13: 93–109, 2002. 1.1.1

[52] Gordon M. Shepherd, editor. *The Synaptic Organization of the Brain.* Oxford University Press, 198 Madison Avenue, New York, New York, 5 edition, 2004. ISBN 0-19-515955-1. (document)

[53] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley Longman, Amsterdam, February 2000. ISBN 0201700735. 2.1.2

[54] Mark Summerfield. *Rapid GUI Programming with Python and Qt*. Prentice Hall, 2008. ISBN 0132354187. 2.1.2

[55] David Sussillo, Taro Toyoizumi, and Wolfgang Maass. Self-Tuning of Neural Circuits Through Short-Term Synaptic Plasticity. *J Neurophysiol*, 97(6):4079–4095, 2007. doi: 10.1152/jn.01357.2006. 1.1.1

[56] Bernhard Vogginger. Testing the operation workflow of a neuromorphic hardware system with a functionally accurate model. Diploma thesis, University of Heidelberg, HD-KIP-10-12, http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=2003, 2010. 1.2.2

[57] K. Wendt, M. Ehrlich, and R. Schüffny. Gmpath - a path language for navigation, information query and modification of data graphs. In *Proceedings of the Artificial Neural Networks and Intelligent Information Processing Conference (ANNIIP) 2010*, pages 31–42, 2010. 1.2.4

[58] Karsten Wendt, Matthias Ehrlich, and René Schüffny. A graph theoretical approach for a multistep mapping software for the facets project. In *CEA'08: Proceedings of the 2nd WSEAS International Conference on Computer Engineering and Applications*, pages 189–194, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS). ISBN 978-960-6766-33-6. 1.2.4

Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, 30. August 2010

....
(Unterschrift)