F09/F10 Neuromorphic Computing

Grübl, Andreas
agruebl@kip.uni-heidelberg.de

Baumbach, Andreas
andreas.baumbach@kip.uni-heidelberg.de

May 4, 2021

Revision History

Revision	Date	Author(s)	Description
0.5	10.10.16	AG	created
0.6	13.04.17	AG	correction
1.0	11.10.17	AB, AG	Added XOR-experiment
1.1	13.11.17	AB, AG	Improved readability

This script has mainly been compiled from the following references:
[6, 10, 15]

Contents

1. Introduction **4**

2. Biological Background **5**
2.1. Conductance-based Leaky Integrate and Fire Model 6
2.2. Plasticity 8

3. The Neuromorphic System **10**
3.1. The Neuromorphic Chip 11
3.1.1. Short Term Plasticity (short term plasticity (STP)) 12
3.2. Readout of Analog Signals 12
3.3. System Environment 13
3.4. Configurability 14
3.5. Calibration 15

4. Experiments **16**
4.1. Investigating a Single Neuron 16
4.2. Calibrating Neuron Parameters 18
4.3. A Single Neuron with Synaptic Input 20
4.4. Short Term Plasticity 22
4.5. Feed-Forward Networks 23
4.6. Recurrent Networks 25
4.7. A Simple Computation - XOR 27

A. Acronyms **30**

B. Relevant Technical Configuration Parameters **31**

C. The PyNN Language **32**

D. Linux Basics **38**

Prerequisites This experiment will introduce *neuromorphic hardware* that has been developed in Heidelberg, together with some helpful neuroscientific background. The neuromorphic hardware device, the Spikey chip, is used by means of scripts written in the Python programming/scripting language, which is also used for data analysis and evaluation of the results.

The following Python-based software packages will be used, all are already installed on the computer that will be used for experiment execution:

- Python installation: `Python 2.7.9`
- Generic numerical extension: `numpy 1.8.2`
- Generic plotting: `matplotlib 1.4.2`
- Procedural experiment description `PyNN 0.6`
- Experiment data analysis: `elephant 0.3.0`, based on `neo 0.4.0`

The basic usage of these packages is easy to understand and is demonstrated by means of example scripts. These scripts only need slight modifications/extensions in order to obtain your results. However, a basic understanding of how to write a Python program/script is very helpful for this experiment. A very good introductory tutorial can be found at: `http://www.physi.uni-heidelberg.de/Einrichtungen/AP/Python.php`

Parts of the measurements will be done with an oscilloscope of type IDS-1104B. The manual is available on the FP website[1]. Make yourself familiar with its usage; we will use frequency measurement and basic statistics functions.

We try to provide the necessary neuroscientific background in this script. However, looking at some of the referenced literature might be a good idea. Gerstner and Kistler [9] provides a good overview over different neuron models, in this experiment you will use the leaky-integrate-and-fire (Leaky integrate and fire neuron model (LIF)) model with conductance based synapses. There is also the Spikey school[2] which serves as a starting point for external users. Feel free to also use the references given there.

[1] `http://www.physi.uni-heidelberg.de/Einrichtungen/FP/versuche/anleitungen.php`
[2] `https://electronicvisions.github.io/hbp-sp9-guidebook/pm/spikey/spikey_school.html`

1. Introduction

In this experiment you will characterize and use the Spikey-hardware platform. It is part of an emerging field of neuromorphic computing devices and was developed in Heidelberg at the Kirchhoff Institute for physics.

The term neuromorphic engineering was introduced by Carver Mead, in the late 1980s, and describes the development of analog circuits to mimic brain circuitry [13, 22]. These systems are then used to *emulate* biological neural networks as an alternative to simulating those on traditional von-Neumann architecture computers. A human brain uses about 20 watts of power. The energy budget of the whole human body is about 100 watts. This significant cost factor creates an enormous amount of evolutionary pressure to develop an effective information processing system. The motivation behind neuromorphic research and especially behind the physical hardware development is to understand and then use this biological efficiency.

Before we can discuss the Spikey-system that you will later use in section 3, we will give a brief introduction to the biological background that you will need in order to perform the experiment in section 2. section 4 describes your tasks and in Appendix B, Appendix C and Appendix D you can find an overview over hardware parameters, the PyNN-language and some basic Linux commands respectively. Especially these last sections can have outdated or simply wrong information, take it as a starting point in your search for answers rather than a source of ultimate truth. If you find any mistakes please report them to us.

2. Biological Background

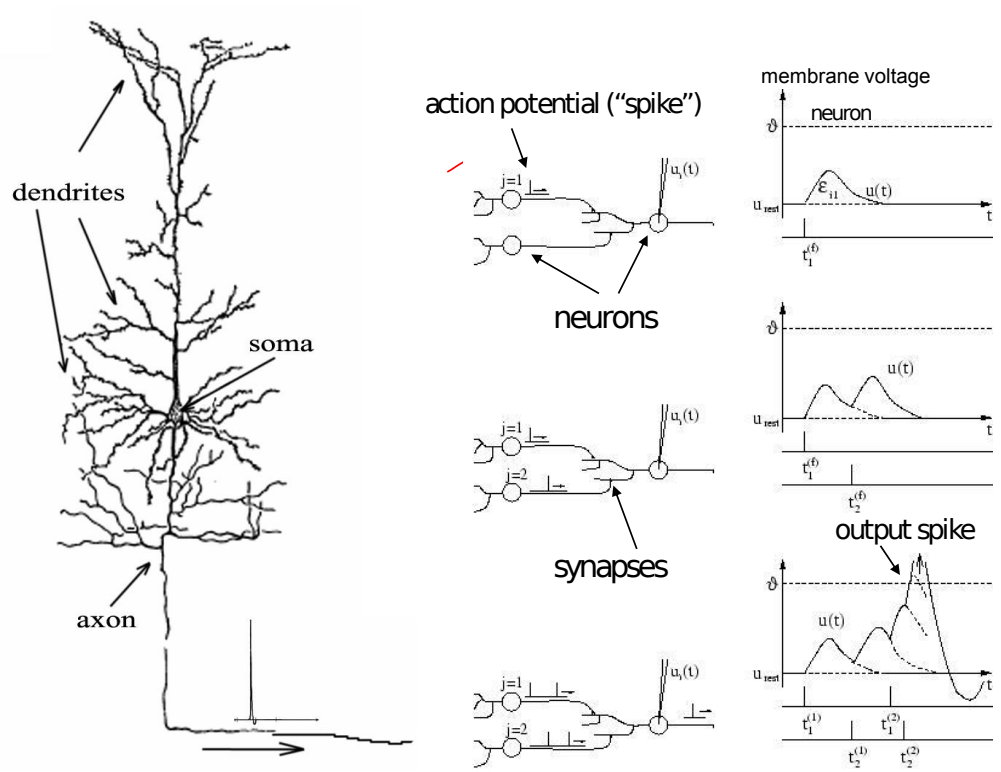


Figure 1: Left: Schematic drawing of a neuron after Ramón y Cajal; dendrites, soma and axon can clearly be distinguished. Figure taken out of [9]. Center: Schematic representation of the neuron circuit, with two presynaptic neurons (left two circles) and one postsynaptic neuron (right circle) with synaptic connections (lines). Right: Schematic membrane voltage time course of the output neuron given the input neurons' behavior. Top: Only one input spike (action potential), one postsynaptic potential (PSP) visible. It forms a difference of exponential, due to the finite membrane time constant. Middle: Both input neurons emit an AP at different times. The output neuron integrates the input and shows the sum of both PSPs. Bottom: Both input neurons emit several APs. The output neuron integrates the input until reaching its threshold potential (θ). At this point a run away effect would take over and the membrane potential would increase dramatically. The output neuron emits an action potential itself. Figure adapted from [1].

The computational power of biological organisms arises from systems of massively interconnected cells, namely neurons. These basic processing elements build a very dense network within the vertebrates' brain (in Latin: *cerebrum*). Most of the cerebral neurons are contained within the *cerebral cortex* that covers parts of the brain surface and occupies an area of about $1.5\,\mathrm{m}^2$ due to its nested and undulating topology. In the human cortex, the neuron density is in the order of 70,000 neurons per cubic millimeter and each neuron receives input from up to 10,000 other neurons, with the connections sometimes spread over very long spatial distances [21].

An overview of a neuron is shown in Figure 1. The typical size of a mammal's neuronal cell body, or *soma*, ranges from 10 to 50 μm and it is connected to its surroundings by a deliquesce set of wires. In terms of information, the *dendrites* are the inputs to the neuron, which has one output, the *axon*. Axons fan out into axonic trees and distribute information to several target neurons by coupling to their respective dendrites via *synapses*.

The voltage over the neuron's membrane (i.e. the difference between the inner-neuron potential and the inter cellar mediums potential) depends on the ion-concentration within it. These are changed actively by ion-pumps and passively by diffusion processes. The former are located at the synapses and can be triggered by activity from other neurons. The activity of the ion pumps also depends on the potential difference on the membrane: If the voltage approaches fast enough a high enough value a run-away effect takes over and leads to a depolarization of the membrane. This dramatic increase in voltage is the so-called *action potential*. Slower gated ion-channels then deterministically pull the neuron's membrane potential back below its equilibrium value (hyper-polarization). See also caption of Figure 1.

The action potential (AP) travels along the axon and activates the synaptic connections along the axonic tree. These trigger ion channels/pumps which modulate the membrane potential of the postsynaptic neurons with a postsynaptic potential (PSP).

The time course of these spikes is hardly dependent on the precise input structure and as such it mainly encodes information in the precise spike time. In this case we remove the depolarization dynamic and instead fix the membrane potential to its hyper-polarization value. This dramatically simplifies our model, while retaining (ideally all) relevant dynamical information.

2.1. Conductance-based Leaky Integrate and Fire Model

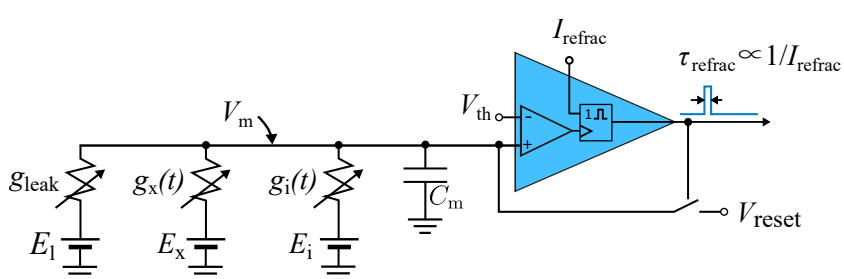


Figure 2: Electronic circuit representation of a leaky integrate and fire neuron model.

For our purposes it is okay to think about the neuron as a capacitor[3] that is connected to multiple voltage sources as depicted in Figure 2. The different ion channels are subsumed to a single excitatory and a single inhibitory potential and the diffusion through the membrane is replaced by the leakage potential. For this simplified model we can write down the governing equation by using Kirchhoff's laws:

$$C_m \frac{dV_{\text{m}}}{dt} = I_C(t) = g_{\text{leak}}(E_{\text{l}} - V_{\text{m}}) + \sum_j g_j^{\text{x}}(t)(E_{\text{x}} - V_{\text{m}}) + \sum_k g_k^{\text{i}}(t)(E_{\text{i}} - V_{\text{m}}) \tag{1}$$

[3] If you are interested in a more complete mathematical description of biologically plausible models you can start with, e.g., [9]

Where C_m is the capacitance of the capacitor, V_m the voltage over it, I_C the current that (de)charges it, g_leak the conductance to the leak potential E_l, and $g_j^\mathrm{x}/g_k^\mathrm{i}$ the conductances to the excitatory/inhibitory so-called reversal potentials $E_\mathrm{x}/E_\mathrm{i}$. This model approximates the behavior of biological neurons without the spiking non-linearity. In order to add this we can simply compare the voltage to a threshold value V_th and say the neuron emits a spike along its axon, triggering all its synaptic connections, when its membrane voltage V_m crosses V_th. In order to also approximate the hyper-polarization at the end of the spike, the neuron enters a *refractory* period in which its potential is clamped to a reset value V_reset. After the refractory time τ_refrac the clamping is released and it continues evolving according to Equation 1. The refractory time τ_refrac can be controlled by a technical parameter I_refrac in Figure 2 which is usesful for our experiment but usually not included in the generic LIF model.

So far we have modeled a single neuron, where we can read back its "output" (i.e., spikes) by the threshold comparison. Interactions between neurons are based on the action potentials traveling down the axonic tree of the presynaptic neuron. The synapses along the axon act as a preprocessor of the arriving information.

This amplitude is the *synaptic strength.* Within the conductance based LIF model, this is implemented as an exponential shaped postsynaptic conductance. This conductance is either to the excitatory or to the inhibitory reversal potential, depending on the presynaptic neuron. This homogeneity is called Dale's law.

With this in mind we can write down the ODE for the conductances $g_j^\mathrm{x}/g_k^\mathrm{i}$:

$$\tau_\mathrm{syn}\frac{dg}{dt} = -g + w\sum\delta(t-t_s) \tag{2}$$

where τ_syn is the synaptic time constant, g is the synaptic conductance, w the synaptic strength and the sum runs over the spike trains of all presynaptic neurons.

Due to the finite membrane capacitance C_m from Equation 1 these exponentials are lowpass-filtered. The resulting trace on the postsynaptic neuron is then a difference of exponentials (also called alpha-shaped) postsynaptic potentials (PSPs). In Figure 3 you can see biological measurements of neurons with relatively short membrane time constants $\tau_m = \frac{C_m}{g_l}$.

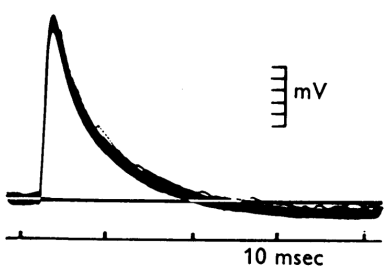


Figure 3: postsynaptic potentials measured in biological tissue (from motoneurons); adapted from [4].

2.2. Plasticity

The synapses also act as a preprocessor of the information arriving at a neuron's dendrite in terms of modulating the effect on the postsynaptic neuron, which is located after the synapse. Modulating in this sense means weighting the presynaptic input in terms of its strength.

The possibility of dynamically changing the synapse weights is called *plasticity*. This change in the effect of a presynaptic signal on the postsynaptic neuron forms the basis of most models of learning and development of neural networks. Spikey implements two forms of plasticity: short term plasticity (Tsodyks Markram Model (TSO)) and spiketime-dependent plasticity (spike timing dependent plasticity (STDP)) [19, 18]. The former modulates the w in Equation 2 as a function of its local history and models resource availability of the neuron. Whereas the latter changes w permanently and models permanent or at least long term connectivity changes. You will only work with the STP circuitry and therefore we will neglect STDP in the following.

Spikey implements a simplified version of Tsodyks-Markram model [19]. Neurotransmitters of a synapse are modeled as being in one of three states (recovered R, effective E and inactive I), whose relations are described by the following equations:

$$1 = R + E + I \tag{3}$$

$$\frac{dE}{dt} = -\frac{E}{\tau_{facil}} + \sum_{spk} UR\delta(t - t_{spk}) \tag{4}$$

$$\frac{dR}{dt} = \frac{I}{\tau_{rec}} - \sum_{spk} UR\delta(t - t_{spk}) \tag{5}$$

The synaptic conductance is proportional to the effective partition E. Essentially $E(t)$ is a multiplicative factor that goes into $g_j(t)/g_k(t)$ in Equation 1.

Starting from a completely relaxed system, i.e. $(R, E, I) = (1, 0, 0)$, an arriving spike transfers a portion $U \in (0, 1)$ (the utilization) of the recovered partition R into the effective partition E. E then decays with time constant τ_{facil} into I and I with time constant τ_{rec} back to R. Different values of τ_{facil} and τ_{rec} lead to different behavior (c.f., Figure 4).

We will discuss its implementation on Spikey in subsubsection 3.1.1.

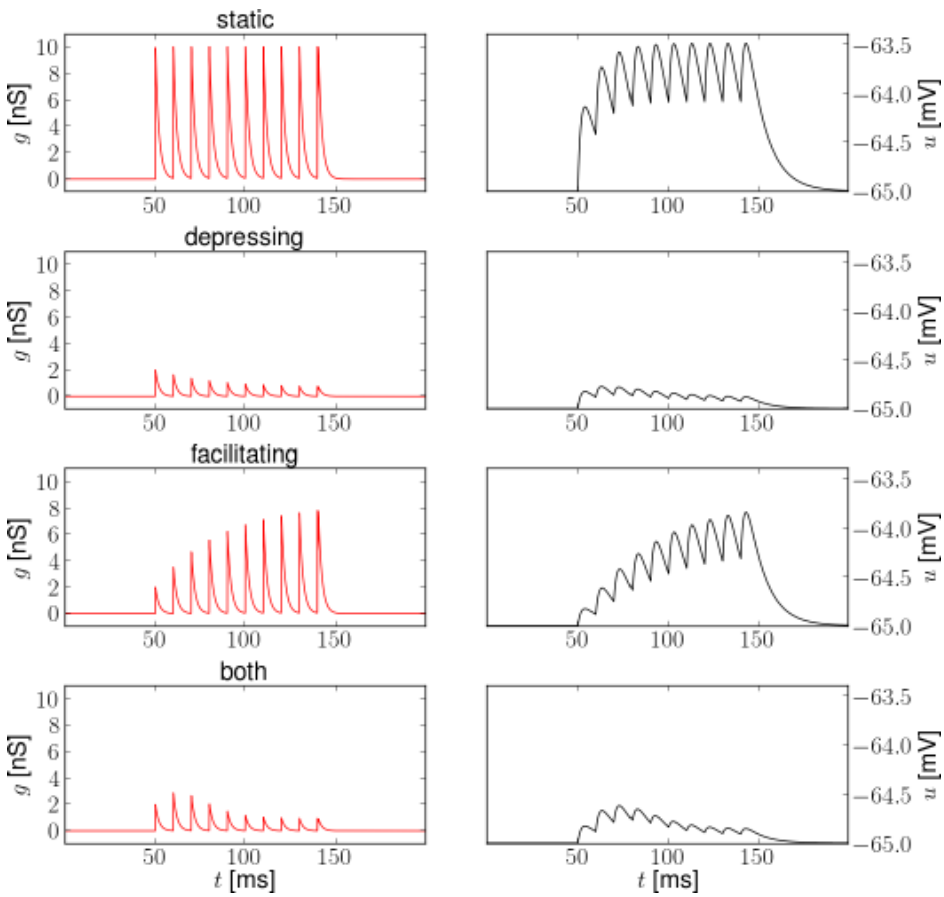


Figure 4: Synaptic conductance (left) and membrane voltage (right) of a neuron under a regular 100 Hz stimulus for different TSO parameters. $U = 0.2$, hence the first PSP is 5 times as high without TSO (top row). $\tau_{rec} = 100\,\text{ms}$ and $\tau_{facil} = 200\,\text{ms}$ when activated, otherwise 0 ms. Figure adapted from [14].

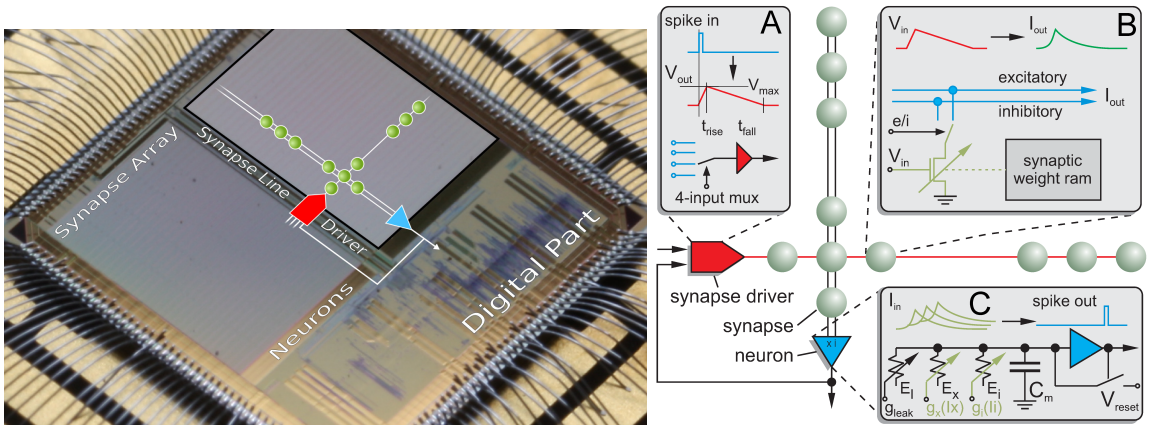


Figure 5: Microphotograph of the *Spikey* ASIC-chip (fabricated in a 180 nm CMOS process with die size $5 \times 5\,\mathrm{mm}^2$). Each of its 384 neurons can be arbitrarily connected to any other neuron. In the following, we give a short overview of the technical implementation of neural networks on the *Spikey* chip. **(A)** Within the synapse array 256 synapse line drivers convert incoming digital spikes (blue) into a linear voltage ramp (red) with a falling slew rate t_{fall}. Each of these synapse line drivers are individually driven by either another on-chip neuron (int), e.g., the one depicted in (C), or an external spike source (ext). **(B)** Within the synapse, depending on its individually configurable weight w_i, the linear voltage ramp (red) is then translated into a current pulse (green) with exponential onset and decay. These postsynaptic pulses are sent to the neuron via the excitatory (exc) and inhibitory (inh) input line, shared by all synapses in that array column. **(C)** Upon reaching the neuron circuit, the total current on both input lines is converted into conductances, respectively. If the membrane potential V_{m} crosses the firing threshold V_{th}, a digital pulse (blue) is generated, which can be recorded and fed back into the synapse array. After any spike, V_{m} is set to V_{reset} for a refractory time period of τ_{refrac}. Neuron and synapse line driver parameters can be configured as summarized in Table 2.

3. The Neuromorphic System

The central component of our neuromorphic hardware system is the neuromorphic microchip *Spikey*. It contains analog very-large-scale integration (very large scale integration (VLSI)) circuits modeling the electrical behavior of neurons and synapses (Figure 5). In such a *physical model*, measurable quantities in the neuromorphic circuitry have corresponding biological equivalents. For example, the membrane potential V_{m} of a neuron is modeled by the voltage over a capacitor C_{m} that, in turn, can be seen as a model of the capacitance of the cell membrane. In contrast to numerical approaches, dynamics of physical quantities like V_{m} evolve continuously in time. We designed our hardware systems to have time constants approximately 10^4 times faster than their biological counterparts allowing for high-throughput computing. This is achieved by reducing the size and hence the time constant of electrical components, which also allows for more neurons and synapses on a single chip. To avoid confusion between hardware and biological domains of time, voltages and currents, all parameters are specified in biological domains throughout this script.

3.1. The Neuromorphic Chip

On *Spikey* (Figure 5), a VLSI version of the standard leaky integrate-and-fire (LIF) neuron model with conductance-based synapses is implemented [7]. The neuromorphic circuits are divided into two blocks containing neurons and the connected synapse array (see Figure 5). Each block contains 192 neuron circuits with 256 synapses per neuron, located in the columns above the neurons. A description of the information flow can be found in the caption of Figure 5.

The time evolution of the membrane potential can be found by Kirchhoff's equations applied to the circuit depicted in Figure 5C:

$$C_{\mathrm{m}}\frac{dV_j}{dt} = g_{\mathrm{leak}}\ (E_{\mathrm{leak}} - V_j) + \sum_n\ w_{j,n}^{max}(t)\ g_n^{\mathrm{x}}(t)\ (E_{\mathrm{x}} - V_j) + \sum_k\ w_{j,k}^{\mathrm{max}}(t)\ g_k^{\mathrm{i}}(t)\ (E_{\mathrm{i}} - V_j) \quad (6)$$

For details on its hardware implementation see Figure 5, [16] and [12].

The constant C_{m} represents the total membrane capacitance. Thus the current flowing on the membrane is modeled multiplying the derivative of the membrane voltage V with C_{m}. The conductance g_{leak} models the ion channels that pull the membrane voltage towards the leakage *reversal potential*[4] E_{l}. The membrane finally will reach this potential, if no other input is present. Excitatory and inhibitory ion channels are modeled by synapses connected to the excitatory and the inhibitory reversal potentials E_{x} and E_{i} respectively. By summing over n, all excitatory synapses are covered by the first sum. The index k runs over all inhibitory synapses in the second sum. So the two sums run over all connected synapses for neuron j.

The synaptic connection strength is composed of two parts, a long-term part w^{max} and a short term part g which are affected by different forms of plasticity (c.f., subsection 2.2). In the conductance-based LIF model, the utilized synaptic conductance of a neuron j is increased whenever a presynaptic partner neuron k spikes and otherwise decays exponentially back to its resting value, which is zero:

$$\tau_{syn}\frac{dg_{j,k}}{dt} = -g_{j,k}(t) + UR(t)\delta(t - t_s) \quad (7)$$

The size of the increase $UR(t)$ gives the utilized fraction of the available synaptic resources at time t. If TSO is deactived then $UR(t) = 1$ and there is no short term dependency. The solution to this differential equation is a sum of exponentially decaying functions, with time constant τ_{syn}. Due to the finite membrane time constant (τ_m) this results in an alpha-shaped PSP on the postsynaptic neuron. This fits the PSP shape observed in biology (c.f., subsection 2.1).

On *Spikey* the synaptic weight w^{max} is implemented as a 4-bit weight which can be modified on longer time scales via spike-time dependent plasticity (STDP). *Spikey* has the capability to apply both STP and STDP, if you are interested [9] is a good place to start, but in this experiment you will only work with STP.

The propagation of spikes within the *Spikey* chip is illustrated in Figure 5 and described in detail by [16]. *Spikes* enter the chip as time-stamped events using standard digital signaling

[4] The reversal potential of a particular ion is the membrane voltage at which there is no net flow of ions from one side of the membrane to the other. The membrane voltage is pulled towards this potential if the according ion channel becomes active.

techniques that facilitate long-range communication, e.g., to the host computer or other chips. Such digital packets are processed in discrete time in the digital part of the chip, where they are transformed into digital *pulses* entering the synapse line driver (blue in Figure 5A). These pulses propagate in continuous time between on-chip neurons, and are optionally transformed back into digital spike packets for off-chip communication.

3.1.1. Short Term Plasticity (STP)

Spikey implements a simplified version of the Tsodyks-Markram mechanism (c.f. subsection 2.2). Unlike TSO, where both τ_{facil} and τ_{rec} may be unequal to zero, Spikey's implementation allows only to set one of the two. With the other being effectively zero. This means we can only produce either facilitating or depressing synapses (2nd and 3rd row in Figure 4) and not a superposition of both (4th row in Figure 4).

In the PyNN-implementation you will have to choose between either setting τ_{facil} or τ_{rec} to zero, otherwise the software will complain. The calibration of the STP time constant is rather imprecise and as such one cannot expect accurate translation of software defined values to the physical realization. Additionally U is implemented as a 2-bit value rather than a floating point number.

For details about the hardware implementation and emulation results, see [17] and Part 5: Short-term plasticity, respectively.

3.2. Readout of Analog Signals

Spike communication and the transmission of chip configuration data to and from the Spikey chip is done via its digital interface. Additionaly, the chip provides 8 analog readout channels that can be used to monitor or record the membrane voltage of selected neurons. All neurons can be connected to these readout channels following a fixed scheme, as illustrated in Figure 6. The connection is established by closing a switch between the respective neuron circuit and the readout channel's signal line.

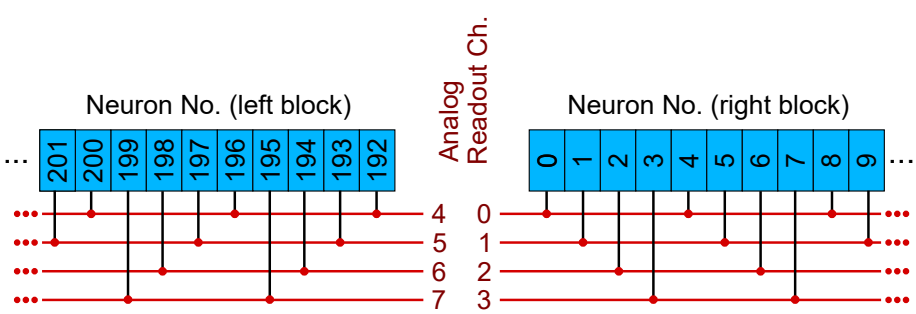


Figure 6: Assignment of physical neuron numbers to analog readout lines on the Spikey chip. Neurons in the left block (i.e. 192 to 383) connect to readout lines 4 to 7; neurons in the right block (i.e. 0 to 191) connect to lines 0 to 3.

Important: You have to ensure that only one neuron connects to each readout channel at a time; i.e. out of neurons 0, 4, 8, ... only one at a time can be recorded via analog readout channel 0. Simultaneously, one out of 1, 5, 9, ... can be connected to channel 1.

The membrane voltage of a neuron can be connected to an analog readout channel with the PyNN-Command `pynn.record_v` (see section Appendix C). This command does a sanity

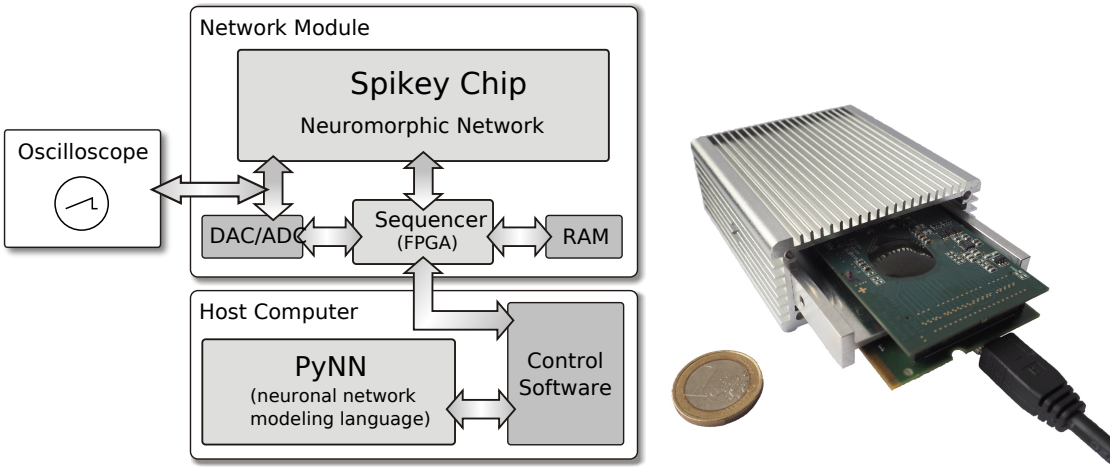


Figure 7: Integrated development environment: User access to the *Spikey* chip is provided using the `PyNN` neural network modeling language. The control software controls and interacts with the network module which is operating the *Spikey* chip. The RAM size (512 MB) limits the total number of spikes for stimulus and spike recordings to approx. $2 \cdot 10^8$ spikes.

check and prints a warning message on the console in case the desired connection cannot be implemented. If only one membrane voltage is to be recorded, it is automatically digitized by an analog-to-digital converter (ADC) that is located on the system. Digitized values can be retrieved using the PyNN-commands `pynn.timeMembraneOutput` and `pynn.membraneOutput`, respectively. Their usage is already given in the Python scripts of the according tasks. If more than one membrane voltage is to be recorded, the ADC is not used, but the voltage outputs are rather only connected to the physical outputs of the chip which are connected to a breakout cable with 8 signal lines. These signals can be connected to an oscilloscope where up to 4 channels can be displayed/recorded, simultaneously.

3.3. System Environment

The *Spikey* chip is mounted on a network module described and shown in Figure 7. Digital spike and configuration data is transferred via direct connections between a field-programmable gate array (field programmable gate array (FPGA)) and the *Spikey* chip. On board digital-to-analog converter (DAC) and analog-to-digital converter (ADC) components supply external parameter voltages to the *Spikey* chip and digitize selected voltages generated by the chip for calibration purposes, or for monitoring of selected membrane voltages. Because communication between a host computer and the FPGA has a limited bandwidth that does not satisfy real-time operation requirements of the *Spikey* chip, experiment execution is controlled by the FPGA while operating the *Spikey* chip in continuous time. To this end, all experiment data is stored in the local random access memory (random access memory (RAM)) of the network module. Once the experiment data is transferred to the local RAM, emulations run with an acceleration factor of 10^4 compared to biological real-time. This acceleration factor applies to all emulations run in this experiment, independent of the size of networks.

During experiment execution, up to 8 membrane voltages of selected neurons can be read out in parallel and be digitized by for example an oscilloscope. The eight analog signals are

connected to a pin header in the *Spikey* USB box which can be connected to the oscilloscope by a flat ribbon cable. This cable is part of the experiment setup.

Execution of an experiment is split up into three steps (Figure 7).

- First, the *control software* within the memory of the host computer generates configuration data (Table 2, e.g., synaptic weights, network connectivity, etc.), as well as input stimuli to the network. All data is stored as a sequence of commands and is transferred to the memory on the network module.
- In the second step, a playback sequencer in the FPGA logic interprets this data and sends it to the *Spikey* chip, as well as triggers the emulation. Data produced by the chip, e.g., neuronal activity in terms of spike times, is recorded in parallel.
- In the third and final step, this recorded data stored in the memory on the network module is retrieved and transmitted to the host computer, where they are processed by the control software.

Having a control software that abstracts hardware greatly simplifies modeling on the neuromorphic hardware system. However, modelers are already struggling with multiple incompatible interfaces to software simulators. That is why our neuromorphic hardware system supports `PyNN`[5], a widely used application programming interface (API) that strives for a coherent user interface, allowing portability of neural network models between different software simulation frameworks (e.g., `NEST` or `NEURON`) and hardware systems (e.g., the *Spikey* system). For details see e.g. [8] for `NEST`, [11] for `NEURON`, [3, 2] for the *Spikey* chip, and [5] for `PyNN`, respectively.

3.4. Configurability

In order to facilitate the emulation of network models inspired by biological neural structures, it is essential to support the implementation of different (cortical) neuron types. From a mathematical perspective, this can be achieved by varying the appropriate parameters of the implemented neuron model Equation 6.

To this end, the *Spikey* chip provides 2969 different analog parameters (Table 2) stored on current memory cells that are continuously refreshed from a digital on-chip memory. Most of these cells deliver individual parameters for each neuron (or synapse line driver), e.g., leakage conductances g_l. Due to the size of the current-voltage conversion circuitry it was not possible to provide individual voltage parameters, such as, e.g., E_l, E_exc and E_inh, for each neuron. As a consequence, groups of 96 neurons share most of these voltage parameters. Parameters that can not be controlled individually are delivered by global current memory cells.

In addition to the possibility of controlling analog parameters, the *Spikey* chip also offers an almost arbitrary configurability of the network topology. As illustrated in Figure 5, the fully configurable *synapse array* allows connections from synapse line drivers (located alongside the array) to arbitrary neurons (located below the array) via synapses whose weights can be set individually with a 4-bit resolution. This limits the maximum fan-in to 256 synapses per neuron, which can be composed of up to 2×192 synapses from on-chip neurons[6], and up

[5] `http://neuralensemble.org/trac/PyNN/wiki/API-0.6`

[6] The chip contains two identical synapse arrays, each driving 192 neurons. Neurons from both halves can drive a synapse driver.

to 256 synapses from external spike sources. Because the total number of neurons exceeds the number of inputs per neuron, an all-to-all connectivity is not possible. For all networks introduced in this experiment, the connection density is much lower than realizable on the chip, which supports the chosen trade-off between inputs per neuron and total neuron count.

In practice most of these parameters are hidden from the user and automatically set according to the user specified PyNN-setup. In this experiment only either the left half (up to neuron number 192) or the right half (starting with neuron number 193) of Spikey will be used.

3.5. Calibration

Device mismatch that arises from hardware production variability causes fixed-pattern noise, which causes parameters to vary from neuron to neuron as well as from synapse to synapse. Electronic noise (including thermal noise) also affects dynamic variables, as, e.g., the membrane potential V_{m}. Consequently, experiments will exhibit some amount of both neuron-to-neuron and trial-to-trial variability given the same input stimulus.

There will always be some level of temporal noise and there isn't much that can be done to remove this completely. However the fixed-pattern neuron-to-neuron noise that originates in the different physical fabrication of the neurons we can compensate for.

Without further treatment we would simply map (by some measure) the user-intended configuration (here done as a PyNN-network) to some hardware configuration using idealized conversion rules. Essentially we would assume that all components are fabricated exactly according to the specification. For example when the user specifies a membrane time constant $\tau_m = \frac{C_m}{g_l} = 10\mathrm{ms}$ it will translated into some value that should implement the required g_l as the capacity C_m is not adaptable. However the parameterized component that controls g_l is not perfectly in accordance to the specification and therefore the realized value of g_l^{real} will be off. This mismatch depends on the chosen neuron, but is stable in time. Therefore we can correct for it by measuring g_l^{real} as a function of the control parameter and build a look-up table to find for a required g_l the necessary parameter value.

This process is called the calibration and reduces the variation seen by the user by more than an order of magnitude. In practice we have a rather good idea of the relation between the parameter and the realized value, only the parameters of this transformation that are unknown. In this case we can, instead of building a look-up table, fit the parameters of the parameter translation.

4. Experiments

4.1. Investigating a Single Neuron

During this task you will learn how to use the *Spikey* neuromophic chip, and how to record relevant analog quantities with an oscilloscope, as well as in software via an analog-to-digital converter (ADC) that is available in the system.

Before you start, make yourself familiar with the hardware setup and establish the necessary connections (ask your supervisor in case anything should be unclear!):

- Power up the Intel NUC computer, your supervisor will set up your credentials. All tasks will be run from this computer.
- Make sure that the Aluminum box with the Spikey chip is connected to the USB hub, and the USB hub to the Intel NUC computer. Turn on the power on the USB-Hub.
- Check the connection of the breakout cable from the pin header in the Aluminum box to the Inputs 1..4 of the Oscilloscope. Connectivity should look like Figure 8:

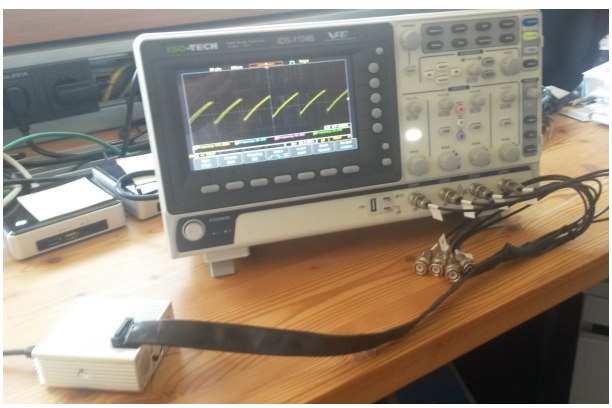


Figure 8: Connect the Aluminum box containing the Spikey system with the oscilloscope using the provided breakout cable

At first, we will investigate the firing behavior of a single neuron without input. The neuron can be brought into a continuous firing regime by setting the leakage reversal potential above the firing threshold (cf., Equation 6).

Tasks:

1. Draw an equivalent circuit of a neuron in the described configuration. Take the neuron schematic in Figure 5 as a reference.

   Question: Which parameters of the neuron will influence to the firing frequency?

2. Go into the folder:

   ```
   ~/fp-spikey
   ```

   Setup your current shell to use Spikey by sourcing the file

```
../env.sh
```

In Appendix D you can find the commands to use, ask your supervisor if you have any questions.

3. Execute the script:

```
experiments/fp_task1_1membrane.py
```

You will see some status output on the terminal.

The script configures one neuron with the described parameters and connects its output to the analog readout lines. The analog signal will be digitized by the on board ADC using the commands

```
pynn.record_v(neuron[0], '')
membrane =pynn.membraneOutput
```

The script also contains example code to plot the membrane voltage trace into the file

```
fp_task1_1membrane.png
```

Look at the generated plot and verify that the values for threshold voltage V_{th} and reset voltage E_{r} are set correctly.

4. In parallel, the membrane can be observed on the oscilloscope. Make yourself familiar with the usage of the oscilloscope and display the membrane voltage output of the configured neuron on channel 1. Determine the average firing rate of the neuron and its standard deviation using the measure and statistics function of the oscilloscope.

   Calculate the mean firing rate of the spikes received in the `spikes` array and compare with the oscilloscope measurement.

   Hint: To obtain a distribution of the interspike intervals (ISIs), you can calculate the pair-wise difference of the received spike times and store them in a new array. The mean value of these differences and its standard deviation can be calculated with the according NumPy functions.

4.2. Calibrating Neuron Parameters

You will already have recognized that the firing rates differ from neuron to neuron. In this experiment you will look at the variations between different neurons. Due to imperfections in the production process the physical circuits (neurons and synapses) show fixed pattern noise. Calibration can reduce this noise as it is approximately constant over time (c.f., subsection 3.5).

During this task you will investigate the variability of the hardware neurons' membrane time constant $\tau_\mathrm{m} = \frac{C_\mathrm{m}}{g_\mathrm{l}}$. It differs from neuron to neuron mostly due to variations in the leakage conductance g_l. The following script sets up 4 identically (in software) configured neurons with parameters that would be valid for an experiment.

```
experiments/fp_task2_calib4membranes.py
```

Tasks:

1. The script already contains valid neuron parameters; according to these parameters, calculate the expected firing rate (cf. Equation 6).

   Note: You can find the `PyNN`-default parameters of the neuron model by using

   ```
   import pyNN.hardware.spikey as pynn
   pynn.IF_facets_hardware1.default_parameters
   ```

   Use this as a reference to see how much you probably deviate from the default. Use Appendix C as a reference for PyNN-commands.

2. Calculate a new firing threshold voltage V_th to bring these neurons into a continuous firing regime for measuring the membrane time constant τ_m, using this equation:

   $$V_\mathrm{th} = E_\mathrm{l} - 1/e \cdot (E_\mathrm{l} - V_\mathrm{reset}) \tag{8}$$

   Given this V_th, a firing rate of $1/(\tau_\mathrm{m} + \tau_\mathrm{refrac})$ is expected. Explain why.

3. Change the setting for V_th in the script accordingly and run the script. Adjust the oscilloscope recording in a way that all 4 membrane voltages can be seen. It should look like Figure 9:

   Use the measurement functions of the oscilloscope to simultaneously measure the firing frequency of the four connected neurons. Enable the statistics function to have the oscilloscope calculate mean values and standard deviation. Note down these values.

4. Calibrate these neurons for identical firing rate, thus identical membrane time constant, by adjusting the leak conductance g_l for each neuron (note down the used values for g_l). Explain possible reasons for the distribution of values, explain how you adapt the g_ls.

5. Investigate the fixed-pattern noise across neurons: Record the firing rates of 192 neurons for the default value of the leak conductance (Note: record all neurons at once). Interpret the distribution of these firing rates by plotting a histogram and calculating the standard deviation. Compare with the plots in Figure 10.

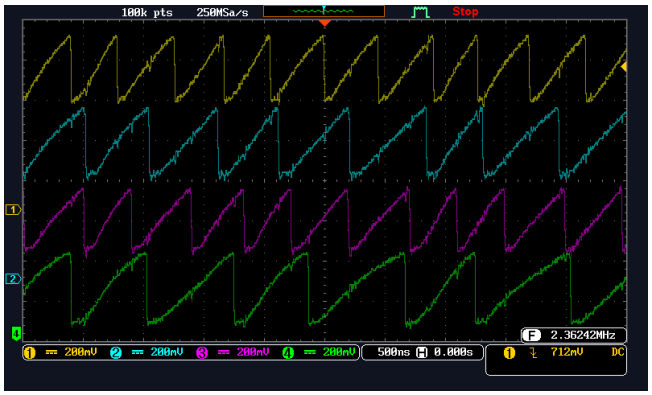


Figure 9: Oscilloscope screen of uncalibrated "identical" neurons.

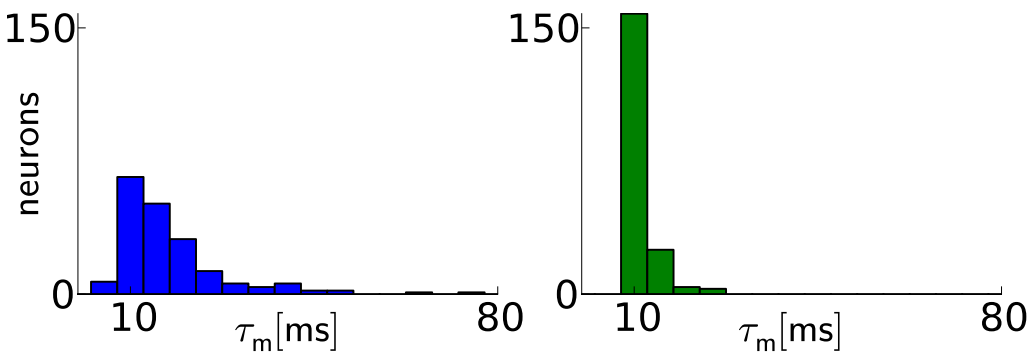


Figure 10: Calibration results for membrane time constants: Before calibration (left), the distribution of τ_m values has a median of $\widetilde{\tau_m} = 15.1\,\text{ms}$ with 20th and 80th percentiles of $\tau_m^{20} = 10.3\,\text{ms}$ and $\tau_m^{80} = 22.1\,\text{ms}$, respectively. After calibration (right), the distribution median lies closer to the target value and narrows significantly: $\widetilde{\tau_m} = 11.2\,\text{ms}$ with $\tau_m^{20} = 10.6\,\text{ms}$ and $\tau_m^{80} = 12.0\,\text{ms}$. Two neurons were discarded, because the automated calibration algorithm did not converge.

6. Optional: Perform a calibration. Automate the tuning that you performed by hand in subtask 4 and attempt to find g_l values such that all 192 neurons have the same membrane time constant. Plot the histogram over the final g_l values and the resulting spike frequencies.

4.3. A Single Neuron with Synaptic Input

In this task, you will evaluate the influence of synaptic input to the neuron. A rather old, nevertheless valid, in vitro measurement of synaptic activity is shown in figure Figure 3. The displayed PSP shows a steep rise from a resting state and an exponential decay back to rest.

In order to reproduce these measurements, you stimulate one hardware neuron with a single synapse and record its membrane potential. The output signal is rather noisy, which is mostly due to the read-out process. In order to average out this part of the signal, the neuron is stimulated with a regular spike train and the spike-triggered average height of these PSPs is calculated. A schematic for the on-chip configuration is shown in figure 11. These measurements will mainly be done using the on-board ADC (see `experiments/fp_task3_synin_epsp.py`), since triggering the EPSPs on the external oscilloscope is tricky. You may try to record the membrane; several EPSPs can be observed, depending on the time scale setting.

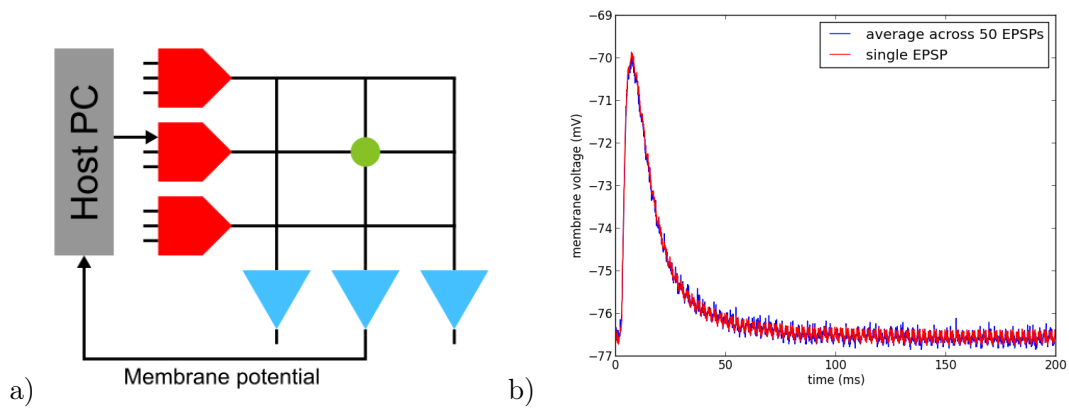


Figure 11: a) A neuron is stimulated using a single synapse and its membrane potential is recorded. The parameters of synapses are adjusted row-wise in the line drivers (red). b) Single and averaged excitatory postsynaptic potentials.

There are two parameters `drvifall` and `drviout` of the synapse line drivers (red box in Figure 11a). Both are technical parameters that influence the PSP shape, that do not have a direct representation in either Equation 6 or Equation 7. Their values can be set from 0 to 2.5. The parameter `drvifall` controls the duration of the falling voltage ramp of the synapse line drivers (smaller values yield a longer ramp! This is not a linear relation!). The parameter `drviout` scales the maximum conductance of the synapse (effectively the realized scale of the four bit value of $w_{j,k}$ in equation Equation 7).

Note: Synaptic weights on hardware can be configured with integer values in the range [0..15] (4-bit resolution). To stay within the range of synaptic weights supported by the hardware, it is useful to specify weights in the domain of these integer values and translate them into biological parameter domain by multiplying them with `pynn.minExcWeight()` or `pynn.minInhWeight()` for excitatory and inhibitory connections, respectively. Synaptic weights that are not multiples of `pynn.minExcWeight()` and `pynn.minInhWeight()` for excitatory and inhibitory synapses, respectively, are stochastically rounded to integer values.

Tasks:

1. Vary the parameters `drvifall` and `drviout` of the synapse line drivers and investigate their effect on the shape of EPSPs (tip: use `pynn.Projection.setDrvifallFactors` and `pynn.Projection.setDrvioutFactors` to scale these parameters, respectively). Make sure that the spike threshold of the neuron is sufficiently large ($> -30\,\mathrm{mV}$) in order to avoid firing. Write down your observations.

2. Compare the PSPs between excitatory to inhibitory synapses.

3. Investigate the fixed-pattern noise across synapses: For a single neuron, vary the row of the stimulating synapse and calculate the variance of the height of the EPSPs across synapses. You need to add an according number of dummy drivers before the actually used synapse driver.

   Plot a histogram over the resulting PSP heights. Make sure that the firing threshold is sufficiently high in order to prevent spiking!

4. In an additional script to this task (`fp_task3_synin_epsp_stacked.py`) you can find commented lines for a different stimulus generation. Use this stimulus to observe stacked EPSPs on the membrane, and reduce the temporal distance between the input spikes until the neuron fires at least once. For this the threshold must not be set to a large value.

   Qualitatively compare the relative heights of the different PSPs with the previous fixed-pattern noise results. Be aware that the input spikes come from different synapse drivers. In light of your results regarding the fixed pattern noise: What do you expect?

5. Optional: Estimate the ratio between fixed-pattern and temporal noise: Measure the distribution over PSP heights in a single run. Write down the mean and the standard deviation for each single synapse. Compare this to the mean and standard deviation of the mean heights of different synapses.

4.4. Short Term Plasticity

In this task, the hardware implementation of Short-term plasticity (STP) is investigated. The circuitry that has been implemented in the Spikey chip reproduces the synapse behavior that is shown in Figure 12. The network description is identical to that shown in Figure 11, but with STP enabled in the synapse line driver.

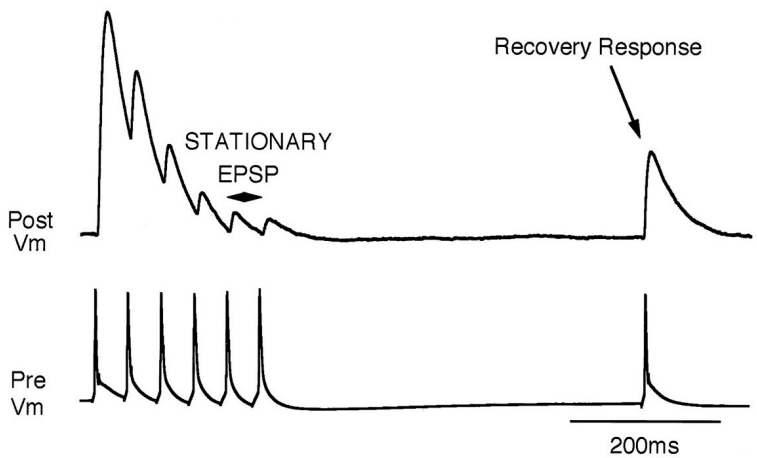


Figure 12: Depressing STP measured in biological tissue (adapted from [20])

.

You can find the required `PyNN`-functions in the script `fp_task4_stp.py`. You may need to tune the synapse parameters a little bit to make the digitized membrane trace similar to Figure 12.

It should be noted that the hardware resolution of utilization U from subsubsection 3.1.1 is limited to 2 bit on *Spikey*. As such there are only 4 different settings of U that can be implemented on chip. Either τ_{rec} or τ_{fac} has to be zero, which selects the STP mode that is currently active (cf. subsubsection 3.1.1). Do not expect these settings to correspond to the exact time constant that will actually be realized.

In general you have to expect a significant amount of trial-to-trial variation in this part of the experiment.

Tasks:

1. Start with the depressing mode ($\tau_{fac} = 0.$). Vary the distance between the initial spikes, and also the distance to the final spike. What do you observe? What happens if you change U? Can you identify the four settings for U? What happens if you change τ_{rec}?

2. Compare the membrane potential to a network with STP disabled.

3. Configure STP to be facilitating and again answer the questions from part 1.

4.5. Feed-Forward Networks

In this task, we learn how to setup networks on the Spikey system. In the last experiments neurons received their input exclusively from external spike sources. Now, we introduce connections between hardware neurons. As an example, a synfire chain with feed-forward inhibition is implemented (for details, see [15]).

The aim is to sustain activity without having to either set the leak potential over the threshold or feeding in external input. For this we will setup a chain of populations of neurons, where each link triggers its down-stream neighbor. After stimulating the first neuron population, network activity propagates along the chain, whereby neurons of the same population fire synchronously. In order to prevent multiple consecutive spikes of a population, each excitatory population has an inhibitory partner population that prevents successive spikes (cf., Figure 13).

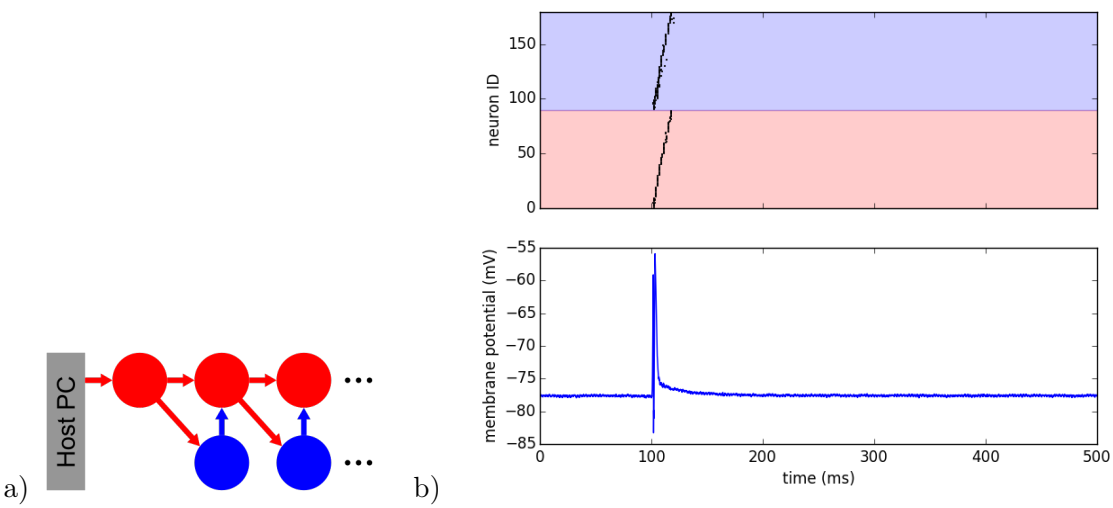


Figure 13: a) Schematic of a synfire chain with feed-forward inhibition. Excitatory and inhibitory neurons are colored red and blue, respectively. b) Emulated network activity of the synfire chain including the membrane potential of the neuron with ID=0 (see example script). The same color code as in the schematic is used. After stimulating the first population, activity propagates through the chain and dies out after the last population.

In PyNN connections between hardware neurons can be treated like connections from external spike sources to hardware neurons. Note again that synaptic weights on hardware can be configured with integer values in the range [0..15] (4-bit resolution). To stay within the range of synaptic weights supported by the hardware, it is useful to specify weights in the domain of these integer values and translate them into biological parameter domain by multiplying them with `pynn.minExcWeight()` or `pynn.minInhWeight()` for excitatory and inhibitory connections, respectively. Synaptic weights that are not multiples of `pynn.minExcWeight()` and `pynn.minInhWeight()` for excitatory and inhibitory synapses, respectively, are stochastically rounded to integer values.

Tasks

1. Tune the weights in the example script to obtain a synfire chain behavior as seen in the Figure 13. Which connection is the most sensitive one? What happens if you disable inhibition?

2. Reduce the number of neurons in each population and use the free neurons to increase the chain size. Which hardware feature limits the minimal number of neurons in each population? What is the maximal chain length that you can produce?

3. Close the loop from the last to the first population (in the python script). Your plots should now look similar to Figure 14). Document your results.

   Record 4 hardware neurons on the oscilloscope from ascending populations to see the temporal difference of the arriving PSPs on the membranes and observe the timing of the arriving excitatory stimulus and the feed-forward inhibition. Convince yourself that the activity is sustained even after the software run completed.

   Tip: For this part it might be easier to switch to a smaller chain with larger populations. Note also that each neuron has a fixed readout-line it can be connected to. Make sure that you record only one neuron per line (see subsection 3.2).

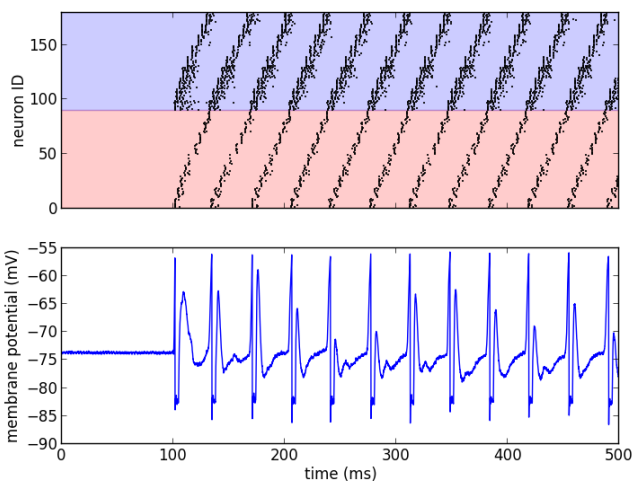


Figure 14: Emulated network activity of the synfire chain in a loop-configuration where the last population stimulates the first population, forming a closed loop.

4.6. Recurrent Networks

In this experiment, a recurrent network of neurons with sparse and random connections is investigated. The purpose of this network will be to add some randomness to an otherwise regularly spiking population of neurons. This network could then be used as a noise source for other networks that are typically modeled in neuroscience. For this "noise" to be useful it needs certain properties.

Typically the utilized noise is produced by a Poisson process. It is beyond the scope of this experiment to discuss the exact properties of such sources, but as a (very crude) measure the coefficient of variation (CV) will be used. For a Poisson process the CV is:

$$CV = \frac{\sigma}{\mu} = 1 \tag{9}$$

where σ is the standard deviation and μ the mean of the produced distribution (in our case of inter spike times (ISI)).

At first, you will set up a population of neurons that uses half of the complete chip without any interconnect, similar to experiment 4.1:

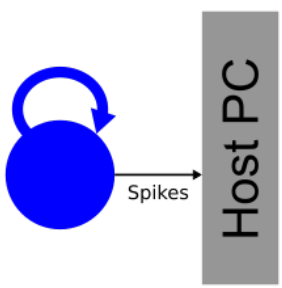
- Set up a population of 192 neurons with standard parameters but the resting potential, e.g.: `neuronParams = {'v_rest': -30.0}` in order to have them in a regular firing regime.
- Record the spike times of all 192 neurons and output the membrane voltage of 4 selected neurons to the oscilloscope, for your "visual" reference. Verify that all neurons that are displayed on the oscilloscope are firing regularly.
- Now activate the inhibitory connections by setting: `active_connections=True`

To avoid self-reinforcing network activity that may arise from excitatory connections, we choose connections between neurons to be inhibitory with weight w. Each neuron is configured to have a fixed number K of presynaptic partners that are randomly drawn from all neurons. These recurrent connections will inhibit the neurons and thereby introduce the irregular spiking behavior. Due to the E_l-over-threshold setting this recurrent network runs hypothetically forever, which allows for easy observations on the oscilloscope.

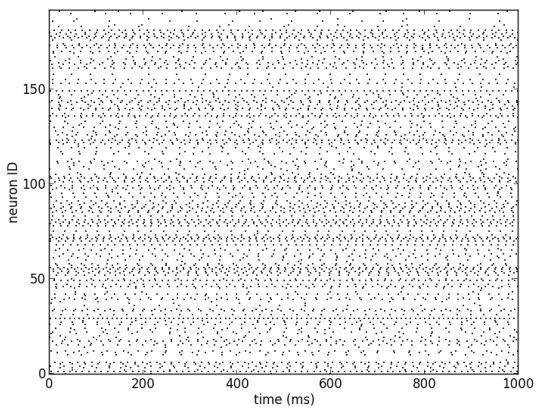
Question: What would happen if we would want to use excitatory connections?

Tasks:

1. With the setup above: For each neuron, measure the firing rate and plot it against the CVs of inter-spike intervals. Interpret the correlation between firing rates and CVs.
2. Measure the dependence of the firing rates and CVs on w and K by sweeping both parameters in your script. Use an appropriate number of w and K values. Visualize the result. You can use the script `fp_task6_plot.py`.
3. Calibrate the network towards a firing rate of approximately 25 Hz. Write down the used parameters and save the results. Optional: Try to maximize the average CV, while keeping the firing rate constant.



a)



b)

Figure 15: a) Schematic of the recurrent network. Neurons within a population of inhibitory neurons are randomly and sparsely connected to each other. b) Network activity of a recurrent network with $K = 15$ (source code lesson 4).

4.7. A Simple Computation - XOR

In the previous tasks you have explored the properties of the *Spikey* neuromorphic chip and have investigated some basic network configurations. Since this experiment is about neuromorphic *computing* we will now look at a network that is not biologically inspired but rather has a technical background. Bit-wise operations (AND, OR, NOT, NAND, XOR and combinations of those) form the basis of traditional binary computations. You will implement a spiking XOR because it is also a standard example of a classification task that is not linearly separable.

A spiking XOR network has the following task: There are two input neurons and one output neuron. If exactly one of the input neurons receives an input the output neuron should fire. Otherwise it should stay silent.

Figure 16 shows one implementation, with excitatory neurons in red and inhibitory neurons in blue. The two inputs (in_1 and in_2) project onto the first layer neurons y_1 and y_2. They each trigger an inhibitory and an excitatory parrot neuron. Where the inhibitory parrots are there to restrict each population to one spike per input spike. The excitatory parrots trigger their respective second layer (h_1 and h_2), while the inhibitory parrots inhibit the opposite second layer (h_2 and h_1 respectively). The second layer neurons also have excitatory and inhibitory parrots, where the excitatory ones also trigger the output neuron.

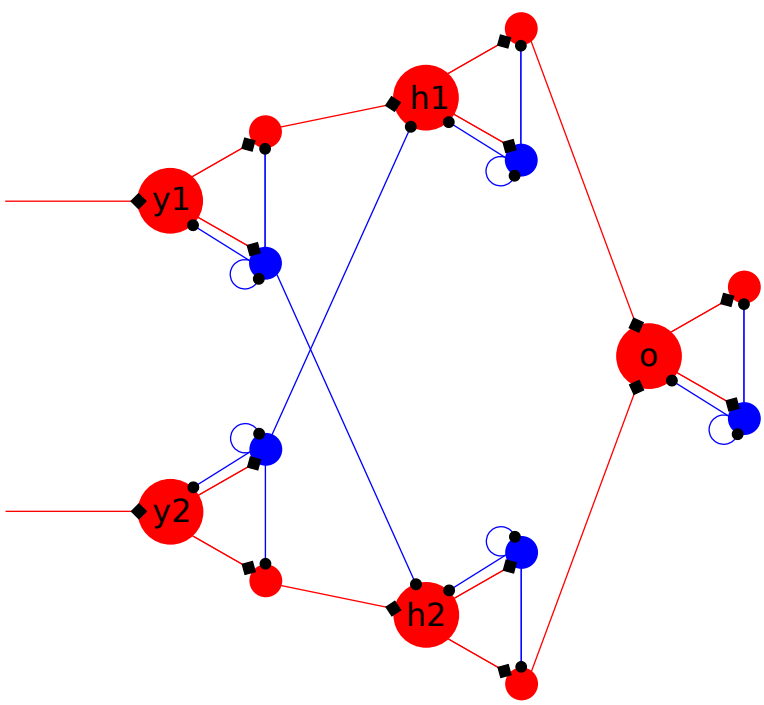


Figure 16: Spiking XOR network

If only one input emits a spike, its side of the network propagates the spike to the output neuron and the inhibitory neurons are only there to limit the number of spikes to one per neuron. If no spike is inserted the network will show no activity at all. And finally if both

inputs emit a spike the inhibitory connections $y_1 \rightarrow h_2$ and $y_2 \rightarrow h_1$ prevent the spike propagation to the output neuron.

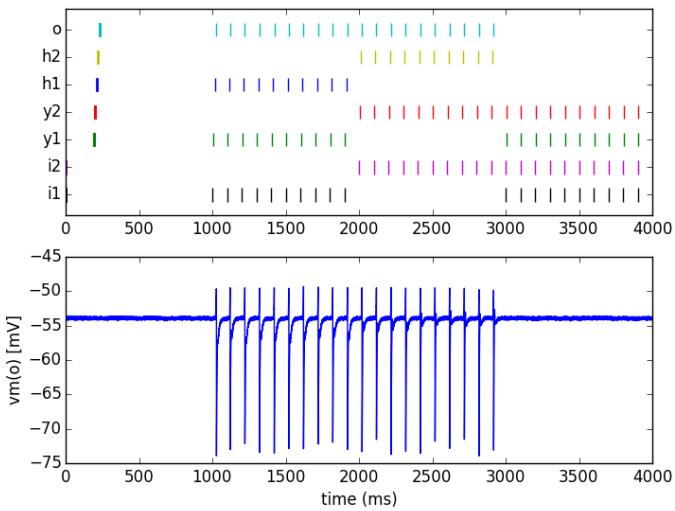


Figure 17: Network activity of a network with XOR functionality.

Tasks:

1. Assume that everything works perfectly (i.e. all neurons work and each input spike triggers exactly one output spike). Can you come up with a smaller network that can perform this task, given that each neuron can only have either excitatory or inhibitory synapses? Draw a network circuit diagram.

2. In the network from Figure 16: Which are the most sensitive connections? How could we make the network more robust? Compare this to subsection 4.5.

3. Find a set of weights I2Yw, Y2Hw, Y2Hi and H2Ow in the script.
   Hint: Find a working point for I2Yw first, then Y2Hw and finally Y2Hi and H2Ow. Can you reproduce the behavior shown in Figure 17?

   Comment: If a population consistently fails to produce the correct behavior you can move it to other physical neurons by adding it to `skip_if_unreliable_list`.

4. What do you expect to happen if there is some jitter on the input? Check the classification rate (correct outputs as a fraction of presented inputs) for different amounts of jitter.

AppendixAppendix

A. Acronyms

AP action potential

ADC analog-to-digital converter

ASIC application specific integrated circuit

CMOS complementary metal oxide semiconductor

DAC digital-to-analog converter

FPGA field programmable gate array

ISI interspike interval

LIF Leaky integrate and fire neuron model

PSP postsynaptic potential

RAM random access memory

STDP spike timing dependent plasticity

STP short term plasticity

TSO Tsodyks Markram Model

VLSI very large scale integration

B. Relevant Technical Configuration Parameters

Scope	Name	Type	Description
Neuron circuits (A)	n/a	$\mathsf{i_n}$	Two digital configuration bits activating the neuron and readout of its membrane voltage
	g_l	$\mathsf{i_n}$	Bias current for neuron leakage circuit
	τ_refrac	$\mathsf{i_n}$	Bias current controlling neuron refractory time
	E_l	$\mathsf{s_n}$	Leakage reversal potential
	E_inh	$\mathsf{s_n}$	Inhibitory reversal potential
	E_exc	$\mathsf{s_n}$	Excitatory reversal potential
	V_th	$\mathsf{s_n}$	Firing threshold voltage
	V_reset	$\mathsf{s_n}$	Reset potential
Synapse line drivers (B)	n/a	$\mathsf{i_l}$	Two digital configuration bits selecting input of line driver
	n/a	$\mathsf{i_l}$	Two digital configuration bits setting line excitatory or inhibitory
	$t_\mathrm{rise}, t_\mathrm{fall}$	$\mathsf{i_l}$	Two bias currents for rising and falling slew rate of presynaptic voltage ramp
	g_i^max	$\mathsf{i_l}$	Bias current controlling maximum voltage of presynaptic voltage ramp
Synapses (B)	w	$\mathsf{i_s}$	4-bit weight of each individual synapse
STP related (C)	n/a	$\mathsf{i_l}$	Two digital configuration bits selecting short-term depression or facilitation
	U_SE	$\mathsf{i_l}$	Two digital configuration bits tuning synaptic efficacy for STP
	n/a	$\mathsf{s_l}$	Bias voltage controlling spike driver pulse length
	τ_rec, τ_facil	$\mathsf{s_l}$	Voltage controlling STP time constant
	I	$\mathsf{s_l}$	Short-term facilitation reference voltage
	R	$\mathsf{s_l}$	Short-term capacitor high potential

Table 2: List of analog current and voltage parameters as well as digital configuration bits. Each with corresponding model parameter names, excluding technical parameters that are only relevant for correctly biasing analog support circuitry or controlling digital chip functionality. Electronic parameters that have no direct translation to model parameters are denoted *n/a*. The membrane capacitance is fixed and identical for all neuron circuits ($C_\mathrm{m} = 0.2\,\mathrm{nF}$ in biological value domain). Parameter types: (i) controllable for each corresponding circuit: 192 for neuron circuits (denoted with subscript n), 256 for synapse line drivers (denoted with subscript l), 49152 for synapses (denoted with subscript s), (s) two values, shared for all even/odd neuron circuits or synapse line drivers, respectively, (g) global, one value for all corresponding circuits on the chip. All numbers refer to circuits associated to one synapse array and are doubled for the whole chip. For parameters denoted by (A) see Equation 6 and [16], for (B) see Figure 5 and [7], for (C) see [17].

C. The PyNN Language

C.1. Basic Commands

Basic commands, required for every simulation

C.1.1. pynn.setup

Has to be called in the beginning of any experiment.

```
pynn.setup(mappingOffset =2)
```

The parameter *mappingOffset* gives the physical neuron ID where the mapping starts.

C.1.2. pynn.Population

Constructs a population (group) of neurons.

```
pynn.Population(dims=5, cellclass =pynn.IF_facets_hardware1,
               cellparams =params, label ='')
```

The parameter *dims* specifies the number of neurons in the population. The neurons will be placed sequentially starting with neuron ID *mappingOffset* (see pynn.setup). Additional population will be placed directly behind the last constructed population.
cellclass specifies the neuron model, in this case always Spikey neurons (*pynn.IF_facets_hardware1*).
cellparams contains the configurable parameters of the neuron model, for the format see *pynn.IF_facets_hardware1.default_parameters*.

C.1.3. pynn.Projection

Construct connections between population of neurons.

```
pynn.Projection(presynaptic_population =pop1,
              postsynaptic_population =pop2,
              method =connector, target ='excitatory')
```

The *postsynaptic_population* receives the spikes of the *presynaptic_population*.
method specifies how the single neurons are connected between the populations (see *Connectors*).
target specifies the type of the interaction (excitatory or inhibitory)

C.1.4. pynn.run

Starts the simulation for the specified network (given by populations and projections).

```
pynn.run(simtime=1000.)
```

Executes the simulation for *simtime* milliseconds.

C.1.5. pynn.end

Concludes the simulation. Necessary before *pynn.setup* can be called again. Required for multiple Simulations in a single script.

C.2. Neurons

C.2.1. pynn.IF_facets_hardware1

Neuron type of the Spikey chip.

```
pynn.IF_facets_hardware1
```

parameters are the neuron parameters that are to be implemented in the hardware neuron. The parameters are represented by a dictionary, if none is specified the following default values are used:

C.2.2. pynn.IF_facets_hardware1.default_parameters

```
pynn.IF_facets_hardware1.default_parameters ={
    "e_rev_I": -80.0,
    "g_leak": 20.0,
    "tau_refrac": 1.0,
    "v_reset": -80.0,
    "v_rest": -75.0,
    "v_thresh": -55.0
}
```

e_rev_I sets the inhibitory reversal potential in millivolt.
g_leak sets the leak conductance *V_rest*.
tau_refrac sets the refractory time in milliseconds.
v_reset sets the reset potential in millivolt.
v_rest sets the rest potential in millivolt.
v_thresh sets the firing threshold in millivolt.
Note: Check the exact spelling. If you have a typo, the default value will silently be used!

C.3. Connectors

Connectors implement connection types between populations.

C.3.1. pynn.AllToAllConnector

Connects all neurons of the presynaptic population with all neurons of the postsynaptic population of a projection (see *pynn.Projection*).

```
pynn.AllToAllConnector(allow_self_connections =True, weights =1.0)
```

If *allow_self_connections* is True (instead of False) neurons will be self-connected. This is only relevant if the pre- and postsynaptic population are identical (i.e. a population is being recurrently connected).
weights sets the strength of the connection in nS (nanosiemens) (see *Hardware Weights*).

C.3.2. pynn.FixedNumberPostConnector

Connects each neuron of the presynaptic population with a fixed number of neurons in the postsynaptic population.

```
pynn.FixedNumberPostConnector(n =10, allow_self_connections =True,
                              weights =1.0)
```

n is the number of connections per neuron of the presynaptic population.
The other parameters are used as with *pynn.AllToAllConnector*.

C.3.3. pynn.FixedNumberPreConnector

Connects each neuron of the postsynaptic population with a fixed number of neurons in the presynaptic population.

```
pynn.FixedNumberPostConnector(n =10, allow_self_connections =True,
                              weights =1.0)
```

n is the number of connections per neuron of the postsynaptic population.
The other parameters are used as with *pynn.AllToAllConnector*.

C.3.4. pynn.FromListConnector

```
pynn.FromListConnector(conn_list =[(pre, post, weight, delay), (...), ...])
```

The list *conn_list* contains the information for the connection between the neuron number *pre* of the presynaptic population and the neuron number *post* of the postsynaptic population with weight *weight* and delay *delay*.
pre and *post* are the indices of the neurons in the respective populations (see *pynn.Projection*).
weight is the respective strength (see *Hardware Weights*)
delay is the synaptic delay in milliseconds that spikes are supposed to have. This is not implemented on the Spikey hardware, therefore this always has to be *None*

C.3.5. pynn.OneToOneConnector

```
pynn.OneToOneConnector(weights =1.0, delays =None)
```

This connector connects the 0-th neuron of the presynaptic population to the 0-th neuron of the postsynaptic population, the 1st to the 1st and so on until all neurons of the smaller population are connected. All other neurons remain unconnected.
The other parameters are identical to the other connectors.

C.4. Sources

Sources are external spike sources, that can be used to stimulate neurons. Either an array with fixed spike-times needs to be provided (*pynn.SpikeSourceArray*) or a source which automatically generates spike times has to be used (*pynn.SpikeSourcePoisson*).

C.4.1. pynn.SpikeSourceArray

```
pynn.SpikeSourceArray
paras ={'spike_times': [1, 10, 20]}
```

Generates external spikes at times 1ms, 10ms and 20ms. The spike times can be provided by an numpy-array or a python-list.

C.4.2. pynn.SpikeSourcePoisson

```
pynn.SpikeSourcePoisson(parameters ={'duration': 1000., 'rate': 1.0,
                                     'start': 0.0})
```

Generates, after *start* ms (here 0ms, i.e. simulation begin) spikes, whose inter spike intervals are Poisson distributed with rate *rate* (here 1.0Hz). *duration* sets the time when the source stops providing spikes (here 1000ms).

C.5. Readout

Methods that are to be called to retrieve measured membrane voltages or spike-times of population. Note: There is only one ADC, so at most one membrane voltage can be recorded and the bandwidth for spike time transfer is limited, groups of 64 neurons transmit spikes over a shared bandwidth.

C.5.1. pynn.record

Called to record the spike times of a population. Has to be called before *pynn.run()*.

```
pop =pynn.Population(dims=5, cellclass =pynn.IF_facets_hardware1,
                     cellparams =params, label ='')
pop.record()
```

record is a method of the population class and can be called after construction of a population.

C.5.2. pynn.getSpikes

Called to retrieve the spikes of a population.

```
pop =pynn.Population(dims=5, cellclass =pynn.IF_facets_hardware1,
                     cellparams =params, label ='')
pop.record()
pynn.run(1000.)
spikes =pop.getSpikes()
```

spikes is now a list consisting of tuples (neuron id, spike time), where neuron id is the number of the neuron from the population. If the population consists of only one neuron the neuron id will be identical for all tuple. In this case you can read out the spike times as:

```
spikes =pop.getSpikes()[:,1]
```

If the population consists of more than a neuron and one wants to read the spike times of e.g. the third neuron, one can use the following:

```
spikes =pop.getSpikes()
spikes =spikes[spikes[:,0]==2, 1]
```

C.5.3. pynn.record_v

```
pynn.record_v(pop[i], '')
```

Records the membrane voltage of the i-th neuron of the *Population* pop (e.g. $i = 0$ is the first neuron). **Warning:** On Spikey there is only one ADC, as such only one membrane trace can be digitized per run.

C.5.4. pynn.timeMembraneOutput

```
times =pynn.timeMembraneOutput
```

times contains the times, when the membrane was digitized.

C.5.5. pynn.membraneOutput

```
membrane =pynn.membraneOutput
```

membrane contains the digitized values of the membrane, for the corresponding times see *pynn.timeMembraneOutput*.

C.6. Hardware Weights

Spikey implements 4-bit synaptic weights. Set weights are stochastically rounded. The following parameter are used to give the weights in terms of the minimum or maximum possible hardware-weight. The configurable weights differ for excitatory and inhibitory synaptic weights.

```
connector =pynn.AllToAllConnector(allow_self_connections =True,
                                  weights =2 * pynn.minExcWeight())
pynn.Projection(presynaptic_population =pop1,
                postsynaptic_population =pop2,
                method =connector, target ='excitatory')
```

C.6.1. pynn.maxExcWeight()

The maximum configurable excitatory weight on Spikey.

C.6.2. pynn.maxInhWeight()

The maximum configurable inhibitory weight on Spikey.

C.6.3. pynn.minExcWeight()

The minimum configurable excitatory weight on Spikey.

C.6.4. pynn.minInhWeight()

The minimum configurable inhibitory weight on Spikey.

In the following example the neurons of population pop1 and pop2 are excitatorily connected. Their weight is twice the minimal possible hardware weight (*2 * pynn.minExcWeight*).

C.7. Minimal Example

```
import pyNN.hardware.spikey as pynn
import matplotlib.pyplot as plt

# Setup experiment
pynn.setup(mappingoffset=0)

# Construct Poisson source
# Setup parameters for the source
poisson_parameters ={'duration': 10000., 'rate': 500.0,'start': 0.0}
# Construct the source itself
pop1 =pynn.Population(dims=1, cellclass =pynn.SpikeSourcePoisson,
                      cellparams=poisson_parameters)

# Construct a single neuron with default parameters
pop2 =pynn.Population(dims=1, cellclass =pynn.IF_facets_hardware1)
# Record the membrane trace of this neuron
pynn.record_v(pop2[0])

# Finally connect the spike source to the neuron
# First specify the type of the connection
connector =pynn.AllToAllConnector(allow_self_connections =True,
                                  weights =10 * pynn.minExcWeight())
# Connect the source to the neuron
pynn.Projection(presynaptic_population =pop1,
               postsynaptic_population =pop2,
               method =connector, target ='excitatory')

# After specifying the simulation (populations and projections) execute the run
pynn.run(10000)

# After the simulation read back the membrane
times =pynn.timeMembraneOutput
membrane =pynn.membraneOutput

# Finalize the experiment
pynn.end()

# Here we could start again with
pynn.setup()
```

D. Linux Basics

D.1. Needed commands

Basically you only need to change to the spikey FP folder, execute the python command in the correct shell-environment and access to a text editor.

D.1.1. cd path

Change Directory. Changes the current working directory of your shell to the path you specified

```
cd ~/fp-spikey
```

Here you change into the folder `fp-spikey` in your HOME-directory. In this directory you find a folder `experiments` and the file `env.sh`.

D.1.2. source file

Executes each line in file in the current shell

```
source env.sh
```

Adapts the environment of the current shell such that spikey can be used.

D.1.3. python file

Executes the python script

```
python experiments/fp_taks1_1membrane.py
```

D.1.4. geany file

Opens file in the text editor geany.

```
geany experiments/fp_taks1_1membrane.py
```

You can of course use any text editor you want.

D.2. Further reading

A playful way to test your abilities is e.g. `https://cmdchallenge.com/`

References

[1] Daniel Brüderle. Implementing spike-based computation on a hardware perceptron. Diploma thesis (English), University of Heidelberg, HD-KIP-04-16, 2004.

[2] 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. 3(17), 2009.

[3] Daniel Brüderle, Mihai Petrovici, Bernhard Vogginger, Matthias Ehrlich, Thomas Pfeil, Sebastian Millner, Andreas Grübl, Karsten Wendt, Eric Müller, Marc-Olivier Schwartz, Dan Husmann de Oliveira, Sebastian Jeltsch, Johannes Fieres, Moritz Schilling, Paul Müller, Oliver Breitwieser, Venelin Petkov, Lyle Muller, Andrew P. Davison, Pradeep Krishnamurthy, Jens Kremkow, Mikael Lundqvist, Eilif Muller, Johannes Partzsch, Stefan Scholze, Lukas Zühl, Alain Destexhe, Markus Diesmann, Tobias C. Potjans, Anders Lansner, René Schüffny, Johannes Schemmel, and Karlheinz Meier. A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems. 104:263–296, 2011. doi: 10.1007/s00422-011-0435-9.

[4] J. S. Coombs, J. C. Eccles, and P. Fatt. Excitatory synaptic action in motoneurones. *The Journal of Physiology*, 130(2):374–395, 1955. ISSN 1469-7793. doi: 10.1113/jphysiol.1955.sp005413. URL `http://dx.doi.org/10.1113/jphysiol.1955.sp005413`.

[5] Andrew Davison, Eilif Muller, Daniel Brüderle, and Jens Kremkow. A common language for neuronal networks in software and hardware. *The Neuromorphic Engineer*, 2010. doi: 10.2417/1201001.1712.

[6] Andrew P. Davison, Eric Müller, Sebastian Schmitt, Bernhard Vogginger, David Lester, and Thomas Pfeil. Hbp neuromorphic computing platform guidebook 0.1 - spikey school. Technical report, 2016.

[7] P. Dayan and L. F. Abbott. *Theoretical Neuroscience*. MIT Press, Cambridge, 2001.

[8] J. M. Eppler, M. Helias, E. Muller, M. Diesmann, and M. Gewaltig. PyNEST: a convenient interface to the NEST simulator. 2:12, 2009. doi: doi:10.3389/neuro.11.012.2008.

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

[10] Andreas Grübl. *VLSI Implementation of a Spiking Neural Network*. PhD thesis, Ruprecht-Karls-University, Heidelberg, 2007. URL `http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=1788`. Document No. HD-KIP 07-10.

[11] Michael L. Hines, Andrew P. Davison, and Eilif Muller. NEURON and Python. *Front. Neuroinform.*, 2009.

[12] Giacomo Indiveri, Bernabe Linares-Barranco, Tara Julia Hamilton, André van Schaik, Ralph Etienne-Cummings, Tobi Delbruck, Shih-Chii Liu, Piotr Dudek, Philipp Häfliger, Sylvie Renaud, Johannes Schemmel, Gert Cauwenberghs, John Arthur, Kai Hynna, Fopefolu Folowosele, Sylvain Saighi, Teresa Serrano-Gotarredona, Jayawan Wijekoon, Yingxue Wang, and Kwabena Boahen. Neuromorphic silicon neuron circuits. 5(73), 2011.

[13] Carver Mead. *Analog VLSI and neural systems*. Addison-Wesley, Boston, MA, USA, 1989.

[14] Mihai A. Petrovici. *Form vs. Function: Theory and Models for Neuronal Substrates*. PhD thesis, Heidelberg - Kirchhoff Institute for Physics, 2015.

[15] Thomas Pfeil, Andreas Grübl, Sebastian Jeltsch, Eric Müller, Paul Müller, Mihai A. Petrovici, Michael Schmuker, Daniel Brüderle, Johannes Schemmel, and Karlheinz Meier. Six networks on a universal neuromorphic computing substrate. *Frontiers in Neuroscience*, 7:11, 2013.

[16] J. Schemmel, A. Grübl, K. Meier, and E. Müller. Implementing synaptic plasticity in a VLSI spiking neural network model. In *Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6, Vancouver, 2006. IEEE Press.

[17] Johannes Schemmel, Daniel Brüderle, Karlheinz Meier, and Boris Ostendorf. Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In *Proceedings of the 2007 International Symposium on Circuits and Systems (ISCAS)*, pages 3367–3370, New Orleans, 2007. IEEE Press.

[18] J. Sjöström and W. Gerstner. Spike-timing dependent plasticity. *Scholarpedia*, 5(2):1362, 2010. doi: 10.4249/scholarpedia.1362. revision #151671.

[19] M. Tsodyks and S. Wu. Short-term synaptic plasticity. *Scholarpedia*, 8(10):3153, 2013. doi: 10.4249/scholarpedia.3153. revision #182489.

[20] Misha V. Tsodyks and Henry Markram. The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability. 94:719–723, January 1997.

[21] Christopher S. von Bartheld, Jami Bahney, and Suzana Herculano-Houzel. The search for true numbers of neurons and glial cells in the human brain: A review of 150 years of cell counting. *Journal of Comparative Neurology*, 524(18):3865–3895, 2016. doi: https://doi.org/10.1002/cne.24040.

[22] Wikipedia. Neuromorphic engineering — wikipedia, the free encyclopedia, 2017. URL `https://en.wikipedia.org/w/index.php?title=Neuromorphic_engineering&oldid=803332319`. [Online; accessed 2-October-2017].