Configuration And Calibration Of Synaptic Elements In A Neuromorphic Hardware System

Ioannis Kokkinos

Electronic Vision(s), Kirchhoff-Institut für Physik
Ruprecht-Karls-Universität Heidelberg

Contents

1 Introduction

„The "Electronic Vision(s) Group" at the "Kirchhoff-Institut für Physik" was founded in 1995“ (Heidelberg-University, 2008) . The group's research includes development, production and programming of artificial neural network chips. Within this internship project an automated, spike based method for configuring and calibrating synapse drivers on neuromorphic hardware is acquired.
The internship is supervised by Dr. Daniel Brüderle.

1.1 Motivation

Due to inevitable fluctuations in the production process, synaptic time constants and efficiencies are subject to random variations.
Therefore the goal of this project is to develop a automated method for calibrating the system. The created software collects data and enters it into a database system (not implemented in this project) for later use in experiments.

1.2 Used Resources And Tools

The following section describes the preexisting resources and tools used in this project. For more details on a specific topic the mentioned reference literature is suggested.

1.2.1 FACETS Neuromorphic Hardware System

The used Hardware System was developed within the FACETS research group, where scientists of different domains, such as modeling experts, engineers and experimentalists collaborate.
All performed experiments run on a "Spikey version 4" chip. It is placed on a Nathan board, which is mounted on a backplane with other Nathan boards. The backplane is connected to a host computer trough gigabit ethernet. The following figure 1 illustrates the setup:

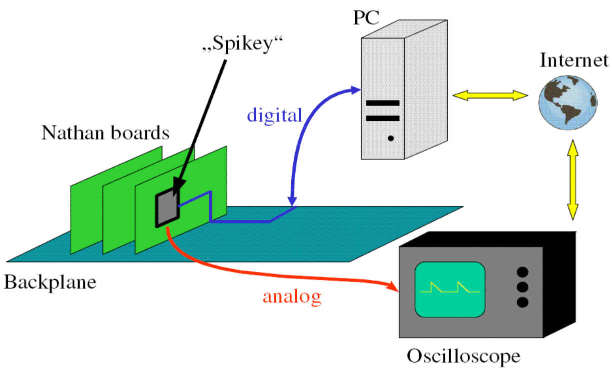

Figure 1: The FACETS stage 1 hardware system. A scope can be used to display psp (post synaptic potential), but it is avoided, as it is much faster to use spike based methods. Further details on the system can be found in Brüderle, 2009.

1.2.2 PyNN

„PyNN (pronounced 'pine') is a simulator-independent language for building neuronal network models" (Davison et al., 2008). It is used to setup experiments and provides many adjustable parameters such as runtime, network size, neuron model, external stimulation input and internal network connections as well as synaptic weights.
The interface is implemented with the script language Python, so it is easy to extend functionality for data evaluation by importing other Python modules, e.g. NumPy for calculations and statistics.

1.2.3 NEST

NEST is a NEural Simulation Tool (Diesmann and Gewaltig, 2002) which, besides the neuromorphic hardware, can be used as a back-end for the PyNN interface. With this tool, experiments and routines can be tested before running them on actual hardware, though performance is not sufficient for large networks.
The major advantage is, that reference experiments can be run on the simulator for comparison with hardware results.

2 Results

In this chapter the methods, the experimental setups and their results are presented.
The general approach is to map biological parameters like the synaptic time constant or the synapse weight to their corresponding hardware parameters, preferably the values of DrviOutBase, DrviFallBase, the 4-bit synaptic weight and, if necessary, the excitatory reversal potential. The values of DrviOutBase an DrviFallBase are not set for each driver, they represent a factor, the individual, driver specific values are multiplied by. With this mapping, an automated (and spike-based) calibration procedure can be implemented.

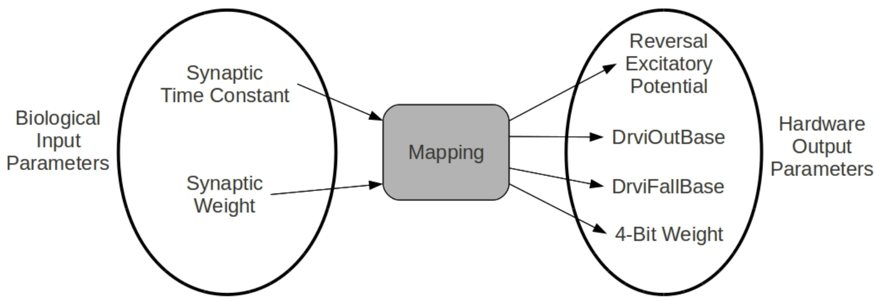

Figure 2: Mapping of input to output parameters. Targeted biological parameters are realized by transforming them into a corresponding hardware setup.

2.1 Configuration And Calibration Methods

The calibration is a complex procedure with many software and hardware specific challenges. The following section will lead through it step by step, while trying to make the underlying thoughts plausible.
The methods presented are spike based, that means that the only feedback available for measurement and control are the output spike trains. Especially the average output frequency will be used to control calibration. This is for two reasons. First, experiments without analog measurement and display on a scope are performed much faster, due to bandwidth limitations between host and scope. Second, a spike based method can be easily transferred to other neuromorphic hardware systems, where access to analog interfaces cannot be guarantied.

2.1.1 Background Stimulation

Every synapse driver can be individually accessed and fed with different input spike trains, so it is possible to use just one driver at a time. But since the synapse drivers are influenced by each other, depending on spiking activity, it would not be a realistic scenario to configure every single driver independently. The neurons can not be calibrated yet, because the process requires already calibrated synaptic drivers. So it is also crucial to average the output firing rate over all available units.
In a first step, a background stimulation is configured. By matching the output firing rate of the hardware with a software reference experiment, comprehensive values for DrviOutBase and DrviFallBase are to found.
This process is applied on each half of available synapse drivers. In that way, two separate background stimulation sources and their corresponding pairs of values for DrviOutBase and DrviFallBase are obtained for further calibration processes. Figure 3 shows the experimental setup of this step.

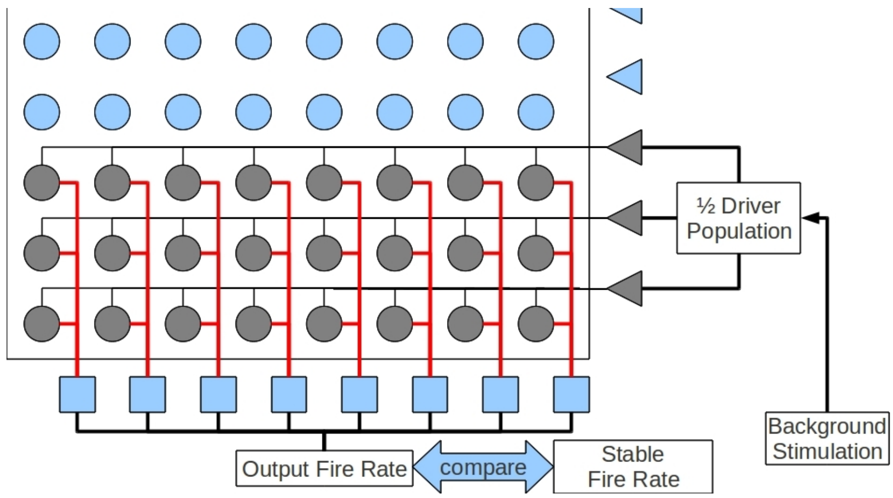

Figure 3: configuring background stimulation. triangular gray: active synapse drivers; circular gray: active synapses; quadratic blue: neurons

For every value of DrviFallBase a value for DrviOutBase can be found, so that it matches the output firing rate of the software simulation at a given input firing rate.
The aim is to find a pair of parameters (or a sweet spot), which does not depend on input frequency.

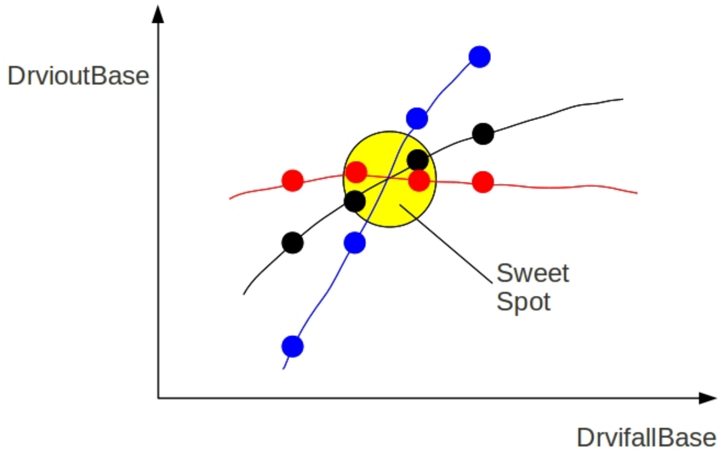

Figure 4: Scan for a parameter independent pair of values. The located sweet spot is the correct hardware configuration, representing the time constant set in software or rather its biological value.

While scanning for the sweet spot, the 4-bit synapse weights are set on their maximum value (15), to facilitate the highest possible resolution. This measurement minimizes later error, caused by modifying synapse weights.

2.1.2 Single Driver Calibration

With the now available background input, the neurons are set in a high conductance state. In this state the output firing rate is very sensitive to variations of the input firing rate. This effect is used for calibrating single synapse drivers under realistic conditions.

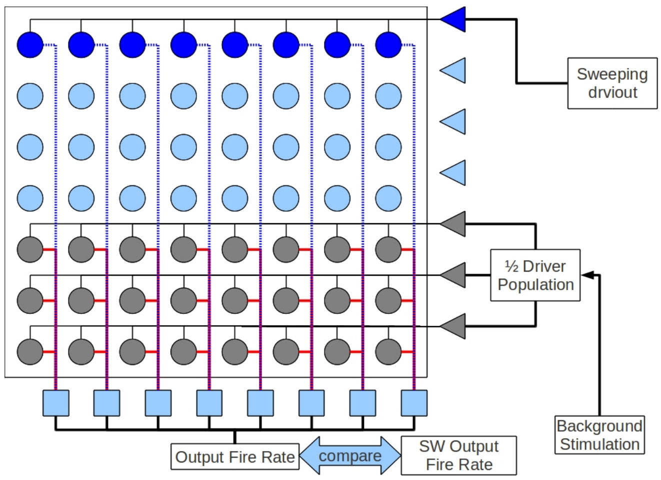

Figure 5: With background stimulation, the other half of the synapse driver population can be calibrated by adapting the single driver specific values for DrviOut and DrviFall to match software simulation.

2.2 Experimental Setups and Results

In this section the series of experiments performed are described and the acquired data presented.

2.2.1 Input Frequency Variation

The first experiment scans the parameter space of DrviOutBase and DrviFallBase for points of equal output firing rate at a given input firing rate.
The input spike trains are poisson distributed, so that the desired stimulation frequency can be adjusted, but the single spikes within the spike train still are uncorrelated. This is necessary for the experiment being based on a realistic stimulation scenario.
For this experiment, a result being in accordance with the considerations made in figure 4 is expected, because the effect of superposition of spikes on output frequency, varies with input frequency. This is due to the specific ratio between input frequency and synaptic time constant.
Despite these considerations, the experimental data shows an equally linear dependency between DrviOutBase and DrviFallBase for the tested input frequencies as can be seen in the plot in figure 6.

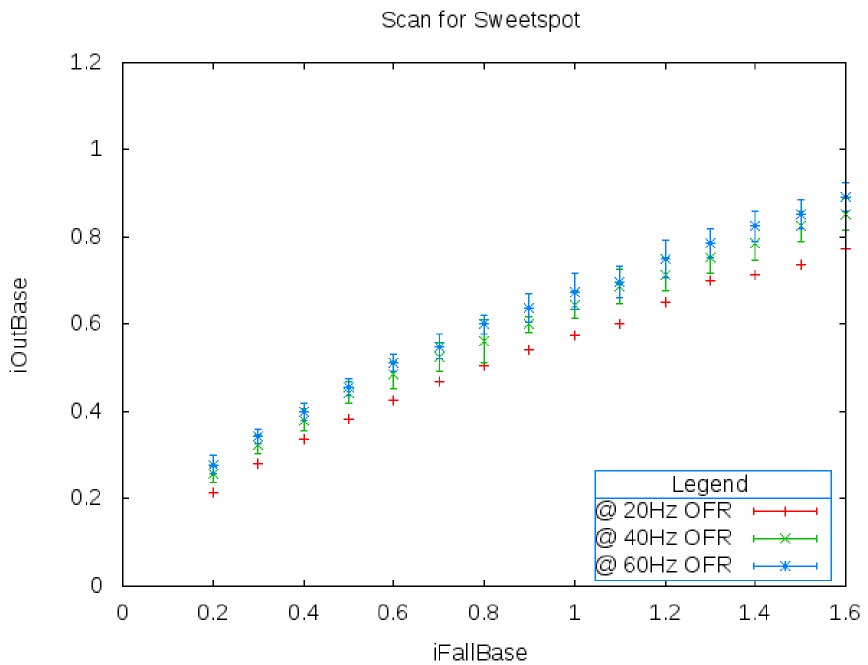

Figure 6: Scan for a sweet spot in the parameter space of DrviOutBase and DrviFallBase. The scan is performed for three different input firing rates between 7 Hz and 12 Hz to match the software output firing rate (OFR) of 20 Hz, respectively 40 Hz, respectively 60 Hz at a given input firing rate. Other parameters are kept constant. Input firing rates above 14 Hz have to be avoided while using 100 or more active synapse drivers, due to input bandwidth limitations.

Interpretations of the plot should be made carefully as there are several possibilities, explaining these results.
One explanation would be, that there are other effects influencing the results, so that the actual effect of superposition remains still remains hidden.
Another possibility is, that the sweet spot lies outside of the scanned parameter space, which can't be extended any further than this.
In both cases the conclusion is, that other parameters have to be tested. Either in order to provoke more impact by the effect of superposition or to translate the position of the sweet spot into range.

2.2.2 Ratio Super Threshold To Sub Threshold

One parameter tested is the ratio of super threshold to sub threshold.

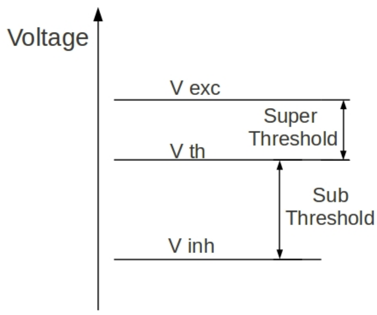

Figure 7: The voltage ratio r arises out of $r = \frac{V_{exc} - V_{th}}{V_{th} - V_{inh}}$. Biological realistic values can be found with $r > 1$. As a hardware parameter the ratio defaults to $r = 0.8$, due to a mechanism enforcing the excitatory potential. This is necessary to support enough dynamic range for the PSP at a low supply voltage.

The scan as introduced in section 2.2.1 is repeated for the values $r = \{0.7, 0.9, 1.2\}$. Figure 8 shows that the ratio has an impact on the results, but only as an offset. The qualitatively behavior remains similar to the measurements with the default value for the ratio.

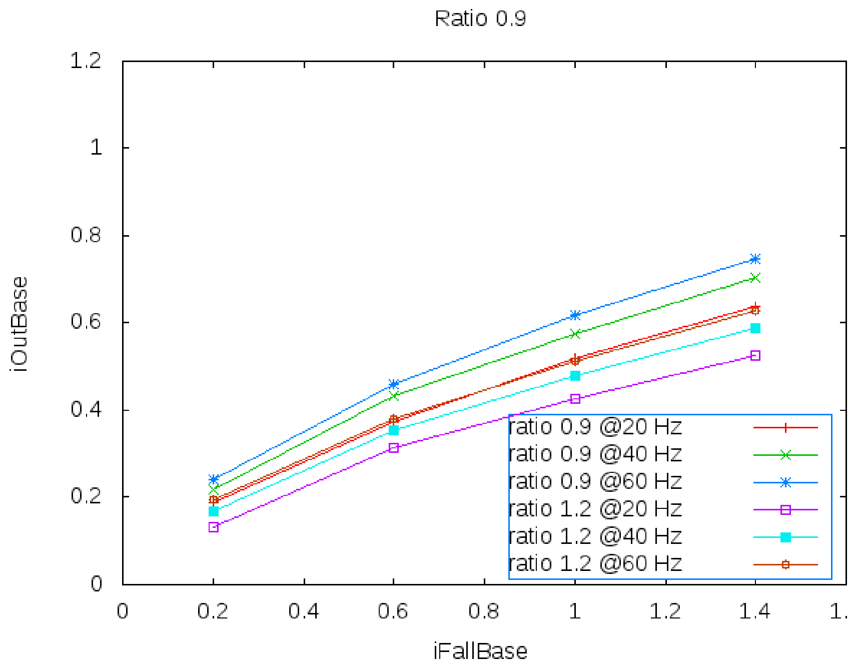

Figure 8: Representative scan for sweet spot at different ratios ($r = 0.9, r = 1,3$). One can see the variation has no qualitative effect on the curve gradient. A higher ratio effectively strengthens the excitatory synapses, so it will result in a lower offset for DrviOutBase.

Taking account of the experimental results, adapting the ratio is not advisable before actually defining constant values for DrviOutBase and DrviFallBase. It should rather be used as a controller to set the optimal working point after configuring the synapse drivers.

2.2.3 Spike Train Cascade

A way of trying to force a visible effect of superposition, is the concept of uniting synapse driver into packages. Within one package every driver has similar input as figure 9 illustrates.
By varying the parameter ΔT the effective input firing rate is controlled. An interval of interest is ΔT as a factor of the biological synapse time constant τ. For $0 \leq \Delta T \leq \tau$ there is strong superposition, while for $\Delta > \tau$ the superposition is reduced to the same level as in the initial experiment in section 2.2.1.
Similar an increase of drivers per package, the number of correlated inputs results in a strong superposition of input, while decreasing the number to one driver per package leads again back to the initial experiment.

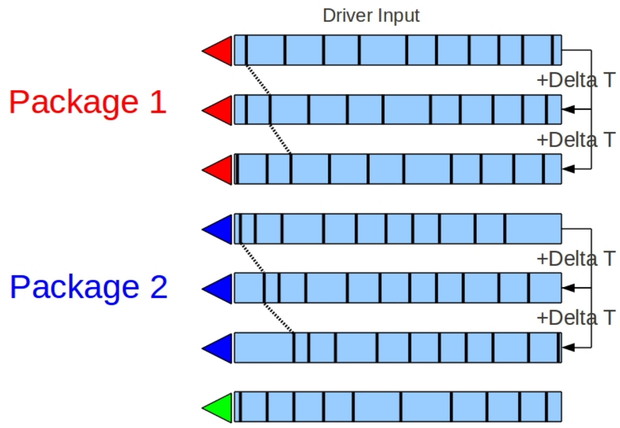

Figure 9: Cascading spike trains. For every package of synapse drivers just one poisson spike train is generated. This spike trains serves as input for every driver within one package, but with different offsets. The first driver of a package receives the original spike train, the next one receives the same spike train, but with an offset of an adjustable parameter ΔT. The next driver input has an offset of $2 \cdot \Delta T$ and so on.

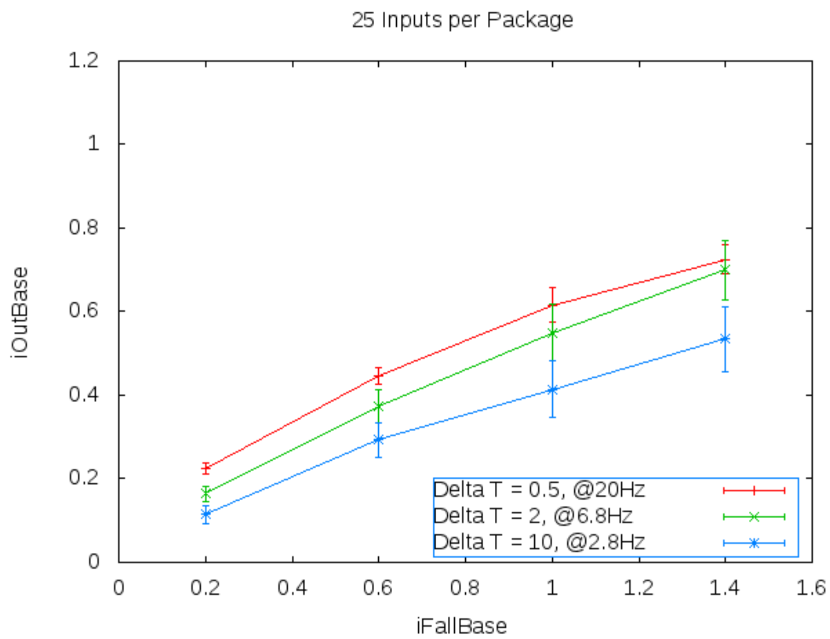

Figure 10: All experiments are setup with 100 synapse drivers.

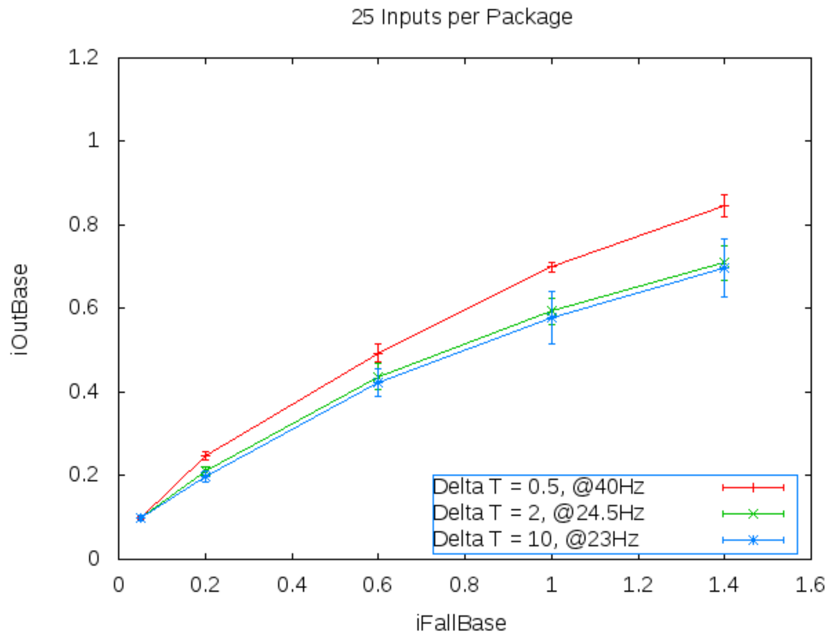

Figure 11: Though one could think the points for different ΔT match for low values of DrviFallBase, this is not the case, because below 0.2 for DrviFallBase, DrviOutBase already takes its minimum value and the target firing rate is not reached.

This method seems to be suited to find a corresponding DrviOutBase DrviFallBase setup for the tested synapse time constant. Though a possible sweet spot is definitely not found, conversion can be observed.
In order to translate the assumed sweet spot into the available parameter range the scan is repeated for other values of τ than 5 ms. But the resulting ratio of input to output firing rate in software simulation is not reproducible by hardware, due to input bandwidth limitations, output bandwidth limitations and high deviation at low firing rates.

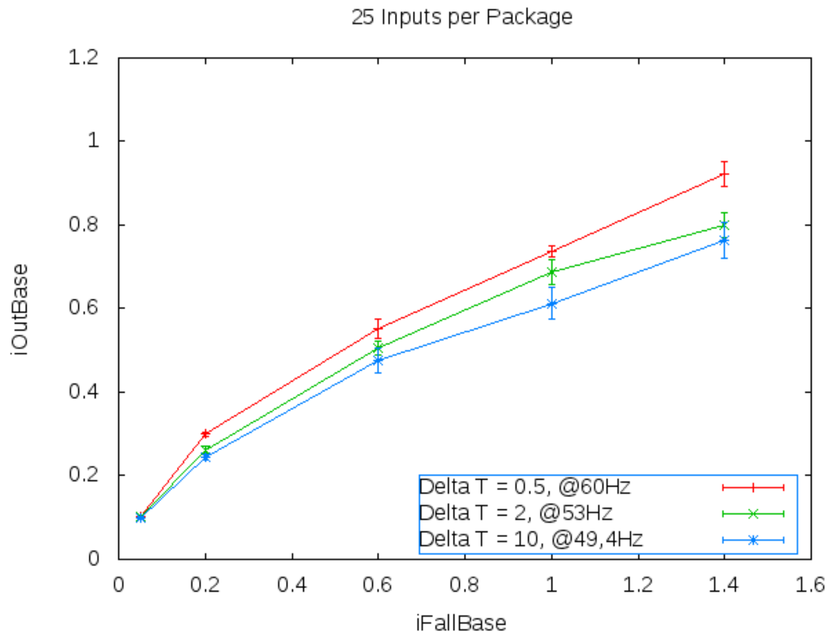

Figure 12: Here the same problem as in figure 11 occurs, below values of 0.2 for DrviFallBase, DrviOutBase already takes its minimum value and the target firing rate is not reached.

3 Discussion

In this section a brief summary points out the most important findings of the project and the vista considers what to challenge next.

3.1 Summary

The initial goal, to collect configuration and calibration data is not reached within this internship, due to problems in configuration and especially in defining a correct setup of the hardware parameters of DrviOutBase and DrviFall Base for a corresponding synaptic time constant.
Nevertheless serviceable data is acquired and mutual experience is gained. In this process the possibilities, available ranges of various parameters and their effect on the synapse drivers are revealed.

3.2 Outlook

The method of cascade spiking, as explained in section 2.2.3, seems to point the way for a solution to configure and finally calibrate the synapse drivers.
Before further testing is performed, current experiments have to be confirmed by scope analyses. Also the source code for the experimental setup should be straightened to be more efficient in order to run more experiments in less time and provide easy access on the top interface level to important variables. It is possible, that the present experiment software contains bugs, which eventually

can be detected this way.
Next next step is to use the configured background stimulation for single synapse driver calibration as proposed in section 2.1.2.
Also the current background stimulation has to be tested for sufficiency. In this case sufficient means, that the output firing rate is stable and sensitive enough to detect additional input from a single driver in order to calibrate it.
The resulting calibration quality has to be analyzed and compared to the uncalibrated system.
With these tasks closed, the system is ready for further calibration.

Appendix

Source Code

The following source code files are written to perform the described experiments. It is build in a modular and object oriented way so that it should be easy to replace single modules or expand functionality.

Listing 1: background.py

```
# script to test various background stimulations with
    different
# iout base and ifall base parameters.
# compare with software simulation.
# by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-
    heidelberg.de
# 12.01.2011

import pyNN.hardware.stage1 as pynnHW
import pyNN.nest as pynnSW

import pylab
import numpy

import poisson_gen
import myrasterplot
import plot
import firerate
import time

# Helper class to store neuron parameters
class NeuronParams:
    def __init__(self,
                 v_reset   = -80.0,
                 e_rev_I   = -75.0,
                 v_rest    = -75.0,
                 v_thresh  = -53.0,
                 g_leak    =  20.0,
                 tau_syn_E=  10.,
                 tau_syn_I=  10.):
```

```
        self.dic = {
                'v_reset'       : v_reset,      # mV
                'e_rev_I'       : e_rev_I,      # mV
                'v_rest'        : v_rest,       # mV
                'v_thresh'      : v_thresh,     # mV
                'g_leak'        : g_leak,       # nS
                'tau_syn_E'     : tau_syn_E,    # ms
                'tau_syn_I'     : tau_syn_I     # ms
        }

# Helper class to store stimulation parameters
class StimParameters:
    def __init__(self,
                 useHardware,                          # Bool
                 ioutBase,                             # Float > 0
                 ifallBase,                            # Float > 0
                 neuronParams,                         # NeuronParams
                 firing_rate_exc,                      # 25 > Float > 0
                 firing_rate_inh,                      # 25 > Float > 0
                 numExcInputs = 20,                    # Int > 0  Exc/Inh~4
                 numInhInputs = -1,                    # Int > 0  Exc/Inh~4
                 spikesRecordPath = 'spikes.dat',      # String
                 statisticsRecordFolder = 'data/',     # String
                 numRuns = 1,                          # Int > 0
                 numNeurons = 2,                       # Int > 1
                 offset = 0,                           # Int >= 0
                 w_excSW = 1.e-16,                     # Float
                 w_inhSW = -1,                         # Float
                 w_exc = 1.0,                          # Float
                 w_inh = -1,                           # Float
                 expDuration = 10000,                  # Int in ms
                 usePlot = False,                      # Bool
                 ratioSupthreshSubthresh = 0.8,        #
```

```
                 deltaTfactor = 1,                # factor is multiplied with tau syn E
                 numCorrInputs = 1                # number of inputs in a correlated input package
                 ):
        self.useHardware  = useHardware
        self.numExcInputs = numExcInputs
        if numInhInputs == -1:
            self.numInhInputs = numExcInputs/1     # !!!! nicht mehr im Verhealtnis 1/4
        else:
            self.numInhInputs = numInhInputs
        self.spikesRecordPath = spikesRecordPath
        self.statisticsRecordFolder = statisticsRecordFolder
        self.numRuns        = numRuns
        if useHardware:
            self.numNeurons    = numNeurons
        else:
            self.numNeurons = 2
        self.offset         = offset
        if useHardware:
            if w_inh < 0:
                self.w_exc          = w_exc
                self.w_inh          = w_exc*0.25
            else:
                self.w_exc          = w_exc
                self.w_inh          = w_inh
        else:
            if w_inhSW < 0:
                self.w_exc          = w_excSW
                self.w_inh          = w_excSW*0.25
            else:
                self.w_exc          = w_excSW
                self.w_inh          = w_inhSW
        self.expDuration  = expDuration
        self.ioutBase     = ioutBase
        self.ifallBase    = ifallBase
        self.neuronParams = neuronParams.dic
        self.firing_rate_exc = firing_rate_exc
        self.firing_rate_inh = firing_rate_inh      # CHANGE BACK TO firing_rate_inh
        self.usePlot      = False
        self.ratioSupthreshSubthresh = ratioSupthreshSubthresh
        self.numCorrInputs = numCorrInputs
        self.deltaT = neuronParams.dic["tau_syn_E"]*deltaTfactor

```

```
# Control and configure the experiment
class Stimulation:
    def __init__(self,
                 stimParameters):
        self.stimParameters  = stimParameters
        self.neuronParams    = stimParameters.neuronParams
        self.ratioSupthreshSubthresh = stimParameters.ratioSupthreshSubthresh
        self.poisson_rng_exc = numpy.random
        self.poisson_rng_exc.seed(int(time.time()*1000))
        self.poisson_rng_inh = numpy.random

    # setup the experiment with given parameters,
    # so that it is runnable.
    # the setup can be changed,
    # all information is stored in class attributes.
    def setup(self, usescope = False, workstationName="station412"):
        if self.stimParameters.useHardware:
            pynnHW.setup(timestep=0.1,
                         debug=False,
                         useScope=usescope,
                         mappingOffset=self.stimParameters.offset,
                         calibOutputPins=False,
                         calibTauMem=False,
                         calibSynDrivers=False,
                         calibVthresh=False,
                         loglevel=0,
                         logfile="logfile",
                         ratioSuperthreshSubthresh = self.ratioSupthreshSubthresh,
                         workStationName=workstationName)
            self.neuron = pynnHW.create(pynnHW.IF_facets_hardware1,
                                        self.neuronParams,
                                        n= self.stimParameters.numNeurons)
            # create empty hardware simulation spike sources
            self.i_exc = pynnHW.create(pynnHW.SpikeSourceArray,
                                       n=self.stimParameters.numExcInputs)
            self.i_inh = pynnHW.create(pynnHW.SpikeSourceArray,
```

```
                                                    n=self.stimParameters.numInhInputs)
        else:
            pynnSW.setup(timestep=0.1)
            self.neuron = pynnSW.create(pynnSW.IF_facets_hardware1,
                                                    self.neuronParams,
                                                    n= self.stimParameters.numNeurons)
            # create empty software simulation spike sources
            self.i_exc = pynnSW.create(pynnSW.SpikeSourceArray,
                                                    n=self.stimParameters.numExcInputs)
            self.i_inh = pynnSW.create(pynnSW.SpikeSourceArray,
                                                    n=self.stimParameters.numInhInputs)
        # fill up with poisson spike trains
        # the inputs are divided into packages,
        # in which every spiketrain is the exact copy of the previous spiketrain,
        # but with a delay of deltaT
        count = 0
        offset = self.stimParameters.deltaT
        for e in self.i_exc:
            if count%self.stimParameters.numCorrInputs == 0:
                # print "package nr " + str(count/self.stimParameters.numCorrInputs + 1)
                newSpikeTrain = poisson_gen.generate(start= 0.0,
                                                    duration= self.stimParameters.expDuration,
                                                    freq= self.stimParameters.firing_rate_exc,
                                                    rng= self.poisson_rng_exc)
            else:
                newSpikeTrain = numpy.array(newSpikeTrain)
```

```
                newSpikeTrain += offset
                for ii in range(len(newSpikeTrain)):
                    if newSpikeTrain[ii] > self.stimParameters.expDuration:
                        newSpikeTrain[ii] -= self.stimParameters.expDuration
                newSpikeTrain.sort()
            # print newSpikeTrain[-1]
            e.set_parameters(spike_times= newSpikeTrain)
            count += 1
        for i in self.i_inh:
            newSpikeTrain = poisson_gen.generate(start= 0.0,
                                                 duration= self.stimParameters.expDuration,
                                                 freq= self.stimParameters.firing_rate_inh,
                                                 rng= self.poisson_rng_inh)
            i.set_parameters(spike_times= newSpikeTrain)
        if self.stimParameters.useHardware:
            # adjust drvifallBase
            pynnHW.hardware.hwa.drvifall_base['exc'] = self.stimParameters.ifallBase
            # adjust drvioutFall
            pynnHW.hardware.hwa.drviout_base['exc'] = self.stimParameters.ioutBase
            # adjust drvifallBase
            pynnHW.hardware.hwa.drvifall_base['inh'] = self.stimParameters.ifallBase
            # adjust drvioutFall
            pynnHW.hardware.hwa.drviout_base['inh'] = self.stimParameters.ioutBase
            pynnHW.connect(self.i_exc,
                           self.neuron,
                           weight= self.stimParameters.w_exc,
                           synapse_type='excitatory')
            pynnHW.connect(self.i_inh,
                           self.neuron,
                           weight= self.stimParameters.w_inh,
                           synapse_type='inhibitory')
            pynnHW.record(self.neuron, self.stimParameters.spikesRecordPath)
```

```
        else:
            pynnSW.connect(self.i_exc,
                           self.neuron,
                           weight= self.stimParameters.
                               w_exc,
                           synapse_type='excitatory')
            pynnSW.connect(self.i_inh,
                           self.neuron,
                           weight= self.stimParameters.
                               w_inh,
                           synapse_type='inhibitory')
            pynnSW.record(self.neuron, self.
                stimParameters.spikesRecordPath)
        if self.stimParameters.usePlot:
            self.rplot = myrasterplot.Rasterplot(self.
                stimParameters.expDuration,
                                                 self.
                                                     neuron
                                                     )

    def resetFiringRates(self):
        self.firingRates = []

    def run(self):
        for i in range(self.stimParameters.numRuns):
            if self.stimParameters.useHardware:
                pynnHW.run(self.stimParameters.
                    expDuration, ratioSuperthreshSubthresh
                     = self.ratioSupthreshSubthresh)
                pynnHW.end()
            else:
                pynnSW.run(self.stimParameters.
                    expDuration)
                pynnSW.end()
            self.firingRates.append(firerate.firerate(
                self.stimParameters.expDuration,
                                                      self
                                                          .
                                                          stimParameters
                                                          .
                                                          numNeurons
                                                          ,

                                                      self
                                                          .
                                                          stimParameters
                                                          .
                                                          spikesRecordPath
                                                          )
                                                          )
```

```
            #print self.firingRates
            if self.stimParameters.usePlot:
                plot.plot(self.stimParameters.spikesRecordPath,
                          self.rplot)

    def statistics(self):
        self.firingRates = numpy.array(self.firingRates)
        fr = firerate.averageFirerate(self.firingRates)
        dr = firerate.sdeviationFirerate(self.firingRates, fr)
        tr = firerate.totalAverage(fr)
        td = firerate.totalDeviation(dr)
        firerate.printToFile(self.stimParameters.statisticsRecordFolder
                             + ' outBase '
                             + str(self.stimParameters.ioutBase)
                             + ' fallBase '
                             + str(self.stimParameters.ifallBase)
                             + '.dat',
                               fr, dr)
        self.tr = tr*1000
        self.td = td*1000
        return self.tr

    def printStatistics(self):
        print "Statistics OFR:"
        print repr(self.tr) + ' ' + '+- ' + repr(self.td)
```

Listing 2: iteration.py

```
# Script to test various background stimulations with different
# iout base and ifall base parameters.
# Compare with software simulation.
# It is build in a modular way,
# so that any component can be replaced easily.
# by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-heidelberg.de
# 26.01.2011

# script module to find the best fitting value


# for given parameters
#    get the sign of a number
#    returns 1 if positive
#    returns 0 if 0
```

```
#     returns -1 if negative
def sign(number):
    if number < 0: return -1
    if number > 0: return 1
    return 0

#     checks if a number is NOT in an intervall
def outOfBound(number, mi, ma):
    return (mi > number or ma < number)


class FitValue:
    def __init__(self,
                 target,          # the value to target
                 minValue,        # minimal value of the variable
                 maxValue,        # maximal value of the variable
                 tolerance = 3.,# difference to target in percent
                 mIterations=10 # max iterations
    #            last Result =? # the last result of the experiment
                ):
        self.target = target
        self.variable = (minValue + maxValue/2.)
        self.tolerance = tolerance
        self.mi = minValue
        self.ma = maxValue
        self.iterations = 0
        self.mIterations = mIterations
    #   self.lastResult = lastResult
    #   NOTE: This last attribute is automatically created
    #   by the following function.

    ###########################################
    # getNewVariable(result)
    ###
    # returns the new value of the variable,
    # returns -1 if there is no better result to expect
    # returns 0 if the target is out of range or reached max iterations
    def getNewVariable(self,result):
        self.iterations = self.iterations +1
        targetAcquired = abs(result-self.target)/self.target < self.tolerance/100.
    #   Oh-Happy-Day-Scenario
        if targetAcquired:
            self.lastResult = result
```

```
            print
            print "target acquired"
            print
            return -1
        if self.iterations > self.mIterations:
            print "reached max iterations"
            return 0
#       check if there has already been a previous result
#       if hasattr(self,'lastResult'):
#           check if (result-target) has same sign as (lastResult-target)
#           if not, we have to turn around and decrease the stepwidth
#           if not(sign(result-self.target)== sign(self.lastResult-self.target)):
#               self.stepwidth = self.stepwidth/2.
#               print "decreasing stepwidth to " + str(self.stepwidth)
#       the result is smaller than the target
        if result < self.target:
            self.mi = self.variable
            self.variable = (self.mi + self.ma)/2.
#       the result is bigger than the target
        else:
            self.ma = self.variable
            self.variable = (self.mi + self.ma)/2.
#       self.lastResult = result
        print
        print "Step #" + str(self.iterations)
        print "New Variable = " + str(self.variable)
        return self.variable

    ###########################################
    # getStepwidth()
    ###
    # Returns the current stepwidth of the iteration.
    # The return value can by interpreted as a max error
    # of the current result.
    def getStepwidth(self):
        return self.ma - self.mi
```

Listing 3: firerate.py

```
# helper functions to get the firerate of neurons
# by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-heidelberg.de
# 08.12.10
# review 22.12.10
# by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-heidelberg.de
# review 11.01.11
```

```python
# by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-heidelberg.de

# review 11.01.11
# changed from decimal to pylab
# import decimal
import pylab as p
import numpy as n

# calculate firerate for every neuron
# review 11.01.11
# load with pylab
def firerate(expDuration,numNeuron,dataPath):
    try:
        spikelist = p.loadtxt(dataPath)
    except:
        return n.array([0.]*numNeuron)
    firelist = []
    #print spikelist
    for i in range(1,numNeuron+1):
        spikes = spikelist[spikelist[:,1]==i]
        firelist.append(float(len(spikes))/expDuration)
    #print firelist
    return n.array(firelist)

# review 11.01.11
# adapted to numpy array
def averageFirerate(firingRates):
    numRuns = len(firingRates)
    #print firingRates
    #print type(firingRates)
    fireList = n.mean(firingRates, axis=0)
    #print fireList
    return fireList

# review 11.01.11
# adapted to numpy array
def sdeviationFirerate(firingRates, avFiringRates):
    devList = n.std(firingRates,axis=0)
    #print devList
    return devList

def printStatistics(avFiringRates, devList):
    print 'Nr Average Firerate      Standard Deviation'
    for i in range(len(avFiringRates)):
        print repr(i).rjust(2), repr(avFiringRates[i]).ljust(35), repr(devList[i]).ljust(35)

def printToFile(fileName, firingRates, devList):
    f = open(fileName, 'w')
```

```
    for i in range(len(firingRates)):
        print >>f, repr(i).rjust(2), repr(firingRates[i]).ljust(20), repr(devList[i]).ljust(22)
    f.close()

# 22.12.2010 total average firingrate calculation
def totalAverage(firerate):
    tr = sum(firerate)/len(firerate)
    return tr

# 17.01.2012 total deviation of firing rate
def totalDeviation(deviation):
    s =0.0
    for i in deviation:
        s= s+i*i
    return (s/len(deviation))**0.5
```

Listing 4: outFallexperiment.py

```
# Class to find the value of ioutBase with given ifallBase,
# to match OFR with software simulation
# by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-heidelberg.de
# 26.01.2011

import background as ba
import iteration as it

class Experiment:
    def __init__(self,
                 ifallBase,        # constant
                 inputs,           # number of exc inputs (inh depending)
                 numRuns,          #
                 numNeurons = 192,#
                 expDuration=6000,#  CHANGE BACK TO 6000
                 usePlot = False,  #
                 ratioSupthreshSubthres = 0.8,
                 deltaTfactor = 1.,
                 numCorrInputs = 1
                 ):
        self.ifallBase  = ifallBase
        self.inputs     = inputs
        self.numRuns    = numRuns
        self.numNeurons = numNeurons
        self.expDuration= expDuration
        self.usePlot    = usePlot
        self.ratioSupthreshSubthresh = ratioSupthreshSubthres
```

```
#       self.refOFR                 # will be created by the SW ref exp
#       self.refIFR                 # will be created by the SW ref exp
        self.w_excSW     = 0.00218
        self.mi          = 0.0
        self.ma          = 1.6
        self.tolerance   = 1.1
        self.mIt         = 8
        self.deltaTfactor  = deltaTfactor
        self.numCorrInputs = numCorrInputs

# Start a software reference experiment with NEST
# the resulting input firing rate should produce the desired
# OFR (output firing rate)
# return a list with ifr, ofr and iterations
    def refExp(self, tfr):
        neuPar = ba.NeuronParams()
        fv = it.FitValue(tfr,                 # target
                         1.,                  # min
                         25.,                 # max
                         tolerance = 1.,      # tolerance
                         mIterations = 40     # max iterations
                        )
        result = 1
        var = fv.variable
        while(var > 0):
            self.refIFR = var
            stiPar = ba.StimParameters(False,
                                       1.,
                                       1.,
                                       neuPar,
                                       var,
                                       var,
                                       numExcInputs = self.inputs,
                                       numRuns = 1,
                                       numNeurons = 2,
                                       w_excSW = self.w_excSW,
                                       expDuration = self.expDuration,
                                       usePlot = self.usePlot,
                                       deltaTfactor = self.deltaTfactor,
                                       numCorrInputs = self.numCorrInputs
```

```
                                )
            stim = ba.Stimulation(stiPar)
            stim.resetFiringRates()
            if fv.iterations < 4:
                for i in range(3):
                    stim.setup()
                    stim.run()
            else:
                for i in range(self.numRuns):
                    stim.setup()
                    stim.run()
            result = stim.statistics()
            stim.printStatistics()
            var = fv.getNewVariable(result)
        self.refOFR = result
        return [self.refIFR, self.refOFR,fv.iterations]

# Start a software reference experiment with NEST
# the resulting output firing rate should be
# reproduceable by hardware
# return a list with ifr, ofr and iterations
    def refDeltaT(self, deltaT):
        self.deltaTfactor = deltaT
        neuPar = ba.NeuronParams()
        result = 1
        stiPar = ba.StimParameters(False,
                                   1.,
                                   1.,
                                   neuPar,
                                   self.refIFR,
                                   self.refIFR,
                                   numExcInputs = self.
                                       inputs,
                                   numRuns = 1,
                                   numNeurons = 2,
                                   w_excSW = self.w_excSW
                                       ,
                                   expDuration = self.
                                       expDuration,
                                   usePlot = self.usePlot
                                       ,
                                   deltaTfactor = self.
                                       deltaTfactor,
                                   numCorrInputs = self.
                                       numCorrInputs
                                   )
        stim = ba.Stimulation(stiPar)
        stim.resetFiringRates()
        for i in range(self.numRuns):
            stim.setup()
```

```
            stim.run()
        result = stim.statistics()
        stim.printStatistics()
        self.refOFR = result
        return result



# Start a experiment with hardware
# the resulting ioutBase should produce the TFR (target firing rate)
    def experiment(self, refOFR):
        neuPar = ba.NeuronParams()
        fv = it.FitValue(refOFR,                      # target
                         self.mi,                     # min
                         self.ma,                     # max
                         tolerance = self.tolerance,  # tolerance
                         mIterations = self.mIt       # max iterations
                        )
        result = 1.
        std = 0.
        var = fv.variable
        while(var > 0):
            self.ioutBase = var
            stiPar = ba.StimParameters(True,
                                       var,
                                       self.ifallBase,
                                       neuPar,
                                       self.refIFR,
                                       self.refIFR,
                                       numExcInputs = self.inputs,
                                       numRuns = 1,
                                       numNeurons = self.numNeurons,
                                       expDuration = self.expDuration,
                                       usePlot = self.usePlot,
                                       deltaTfactor = self.deltaTfactor,
                                       numCorrInputs = self.numCorrInputs
                                      )
            stim = ba.Stimulation(stiPar)
```

```
            stim.resetFiringRates()
            if fv.iterations < 3:
                for i in range(3):
                    stim.setup()
                    stim.run()
            else:
                for i in range(self.numRuns):
                    stim.setup()
                    stim.run()
            result = stim.statistics()
            std = stim.td
            err = fv.getStepwidth()
            var = fv.getNewVariable(result)
            stim.printStatistics()
        return [result, std, fv.iterations, err]

    def getIoutBase(self):
        return self.ioutBase
```

References

Daniel Brüderle. *Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System*. PhD thesis, 2009.

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.

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.

Heidelberg-University. Electronic vision(s) group, http://www.kip.uni-heidelberg.de/cms/groups/vision/, 23.02.2011, 2008.