

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	1
1.1	Motivation	1
1.2	Used Resources And Tools	1
1.2.1	FACETS Neuromorphic Hardware System	1
1.2.2	PyNN	2
1.2.3	NEST	2
2	Results	2
2.1	Configuration And Calibration Methods	3
2.1.1	Background Stimulation	3
2.1.2	Single Driver Calibration	5
2.2	Experimental Setups and Results	6
2.2.1	Input Frequency Variation	6
2.2.2	Ratio Super Threshold To Sub Threshold	7
2.2.3	Spike Train Cascade	8
3	Discussion	11
3.1	Summary	11
3.2	Outlook	11
	Appendix	12
	Source Code	12
	References	27

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:

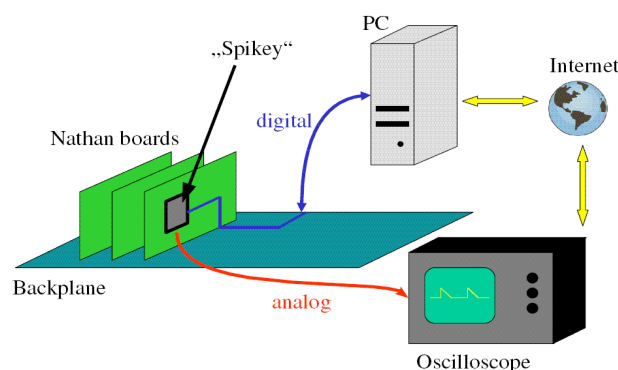


Figure 1: The FACETS stage 1 hardware system. A scope can be used to display psp (post synaptic potential), but it is avoided, as it is much faster to use spike based methods. Further details on the system can be found in Brüderle, 2009.

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 and 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.

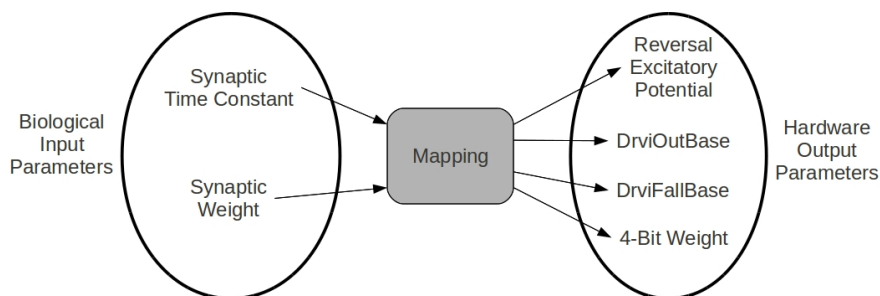


Figure 2: Mapping of input to output parameters. Targeted biological parameters are realized by transforming them into a corresponding hardware setup.

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 guaranteed.

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.

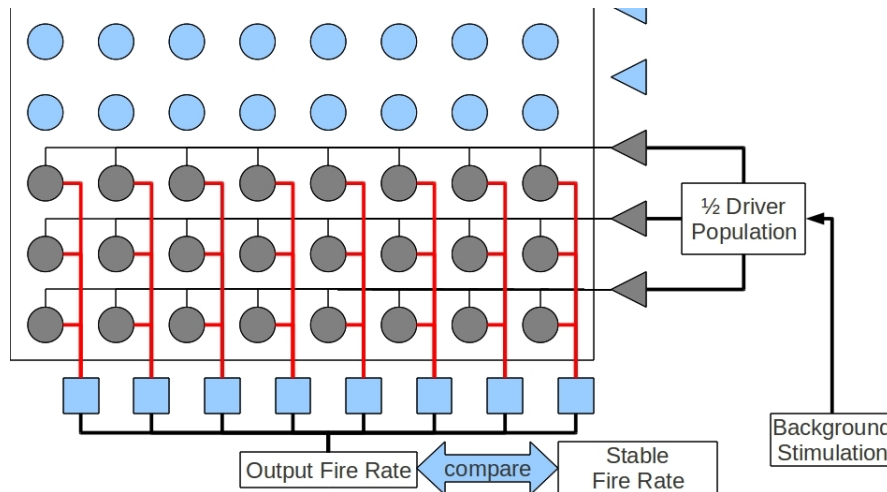


Figure 3: configuring background stimulation. triangular gray: active synapse drivers; circular gray: active synapses; quadratic blue: neurons

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.

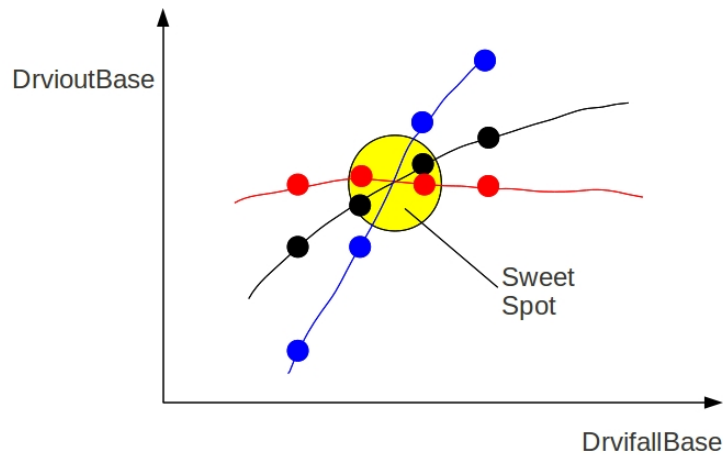


Figure 4: Scan for a parameter independent pair of values. The located sweet spot is the correct hardware configuration, representing the time constant set in software or rather its biological value.

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.

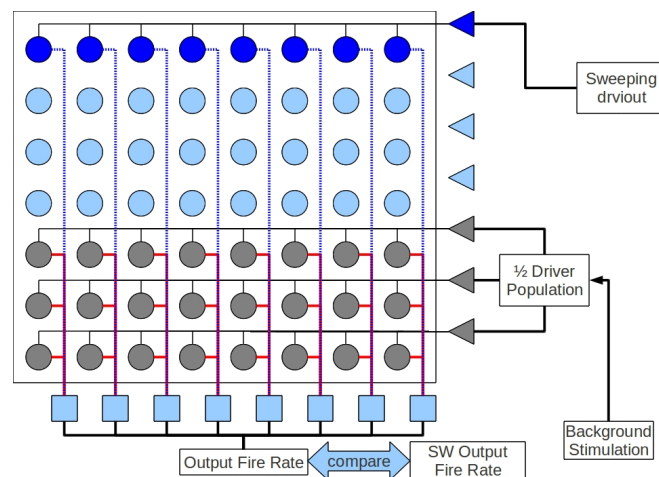


Figure 5: With background stimulation, the other half of the synapse driver population can be calibrated by adapting the single driver specific values for DrviOut and DrviFall to match software simulation.

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.

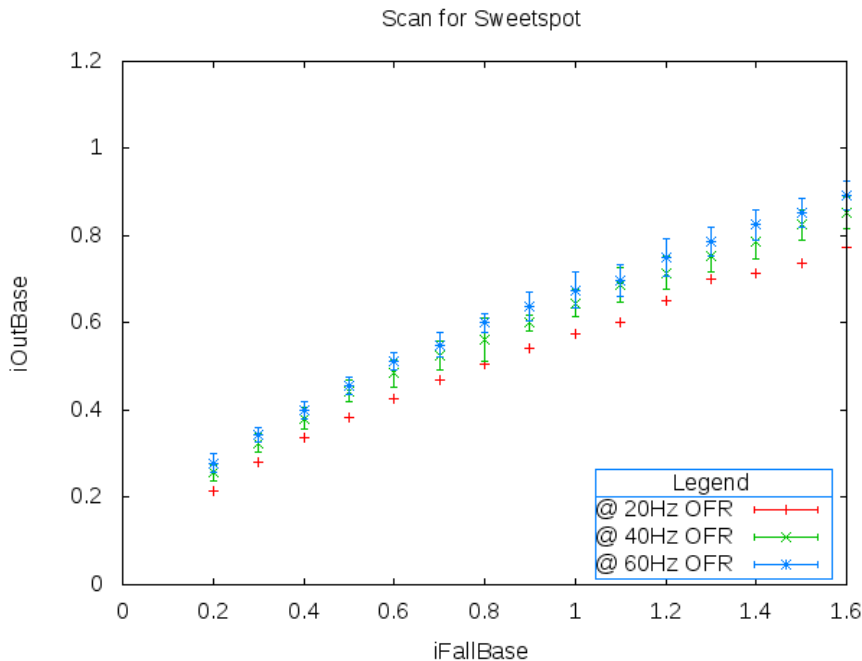


Figure 6: Scan for a sweet spot in the parameter space of DrviOutBase and DrviFallBase. The scan is performed for three different input firing rates between 7 Hz and 12 Hz to match the software output firing rate (OFR) of 20 Hz, respectively 40 Hz, respectively 60 Hz at a given input firing rate. Other 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 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.

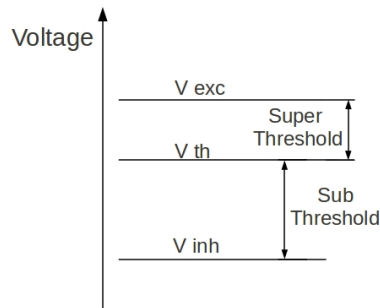


Figure 7: The voltage ratio r arises out of $r = \frac{V_{exc} - V_{th}}{V_{th} - V_{inh}}$. Biological realistic values can be found with $r > 1$. As a hardware parameter the ratio defaults to $r = 0.8$, due to a mechanism enforcing the excitatory potential. This is necessary to support enough dynamic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

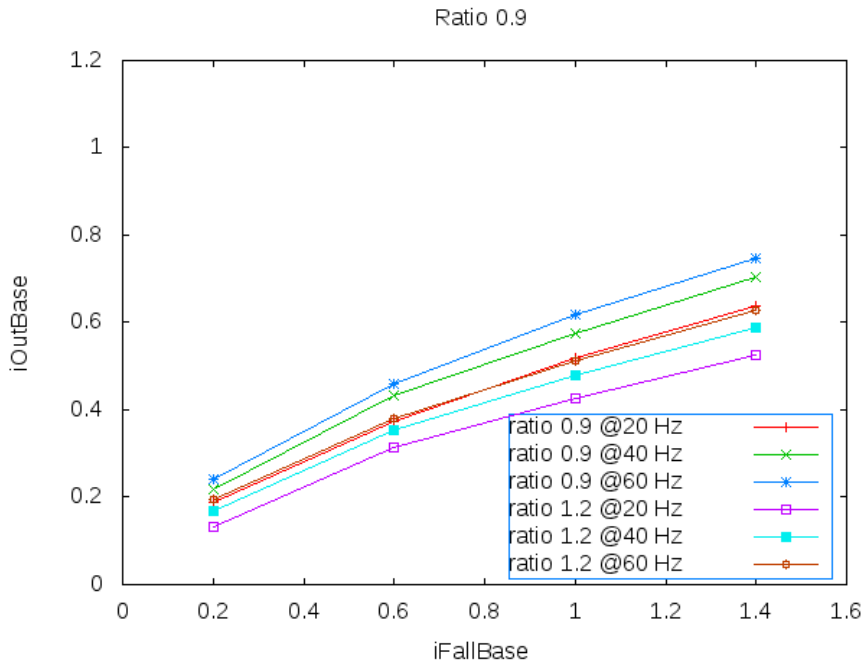


Figure 8: Representative scan for sweet spot at different ratios ($r = 0.9, r = 1, 3$). One can see the variation has no qualitative effect on the curve gradient. A higher ratio effectively strengthens the excitatory synapses, so it will result in a lower offset for DrviOutBase.

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.

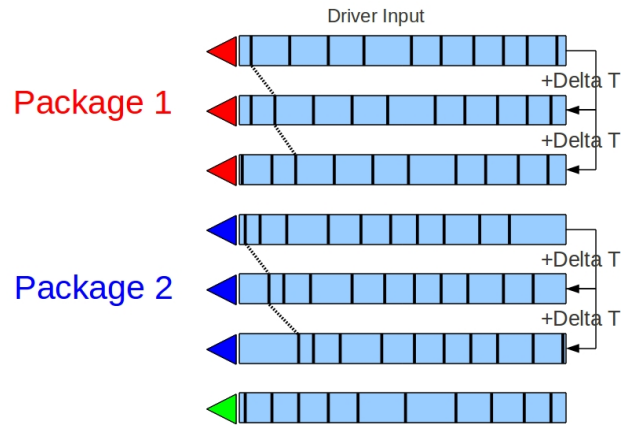


Figure 9: Cascading spike trains. For every package of synapse drivers just one poisson spike train is generated. This spike trains serves as input for every driver within one package, but with different offsets. The first driver of a package receives the original spike train, the next one receives the 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.

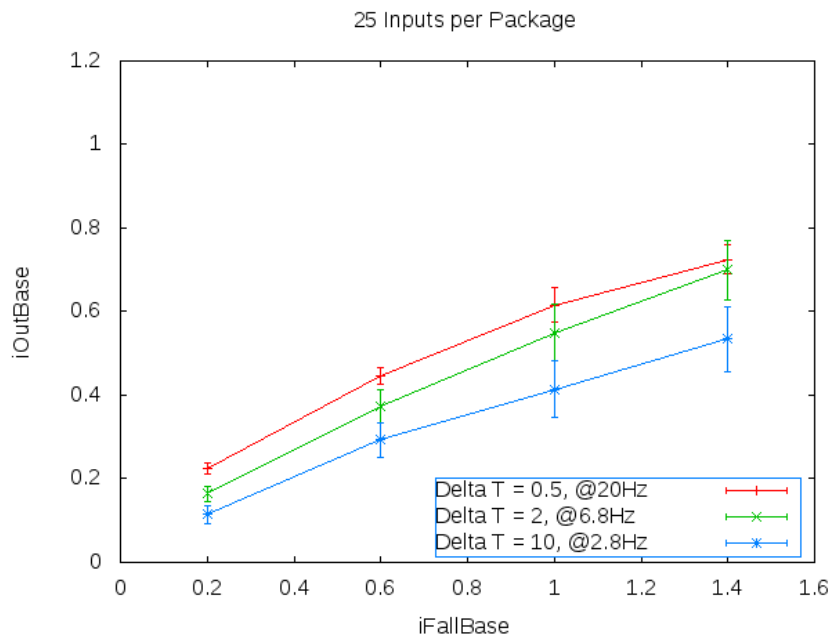


Figure 10: All experiments are setup with 100 synapse drivers.

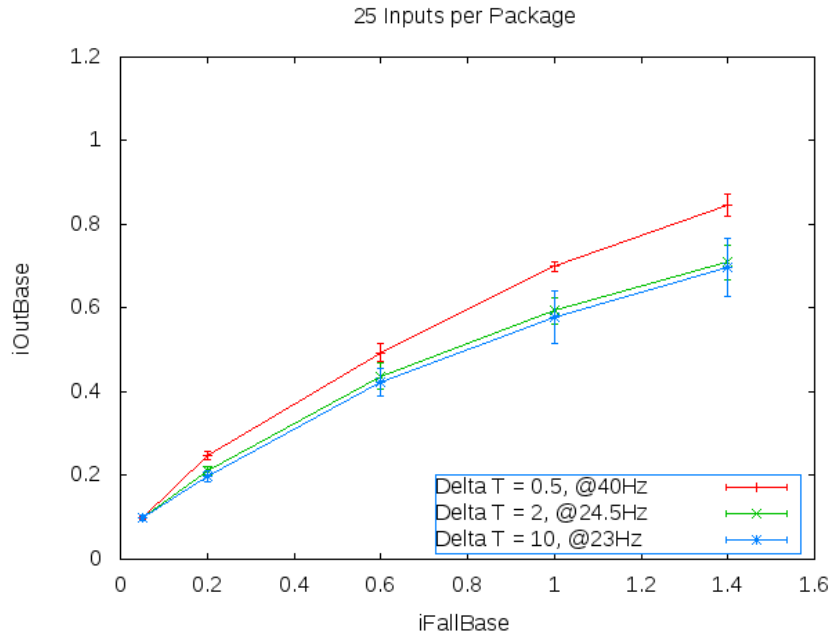


Figure 11: Though one could think the points for different ΔT match for low values of DrviFallBase , this is not the case, because below 0.2 for DrviFallBase , DrviOutBase already takes its minimum value and the target firing rate is not reached.

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.

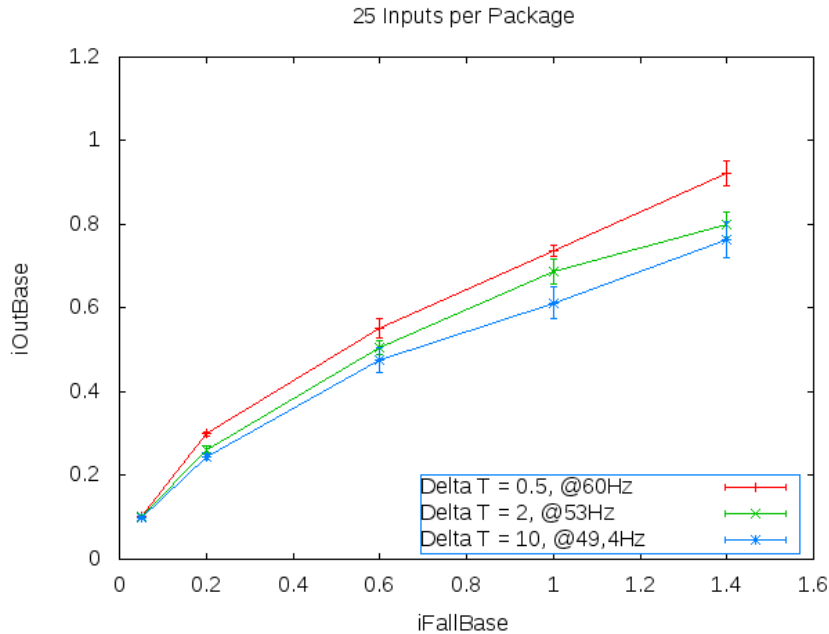


Figure 12: Here the same problem as in figure 11 occurs, below values of 0.2 for $DrviFallBase$, $DrviOutBase$ already takes its minimum value and the target firing rate is not reached.

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 $DrviFallBase$ 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

```
1 # script to test various background stimulations with
   # different
2 # iout base and ifall base parameters.
3 # compare with software simulation.
4 # by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-
   # heidelberg.de
5 # 12.01.2011
6
7 import pyNN.hardware.stage1 as pymmHW
8 import pyNN.nest as pymmSW
9
10 import pylab
11 import numpy
12
13 import poisson_gen
14 import myrasterplot
15 import plot
16 import firerate
17 import time
18
19 # Helper class to store neuron parameters
20 class NeuronParams:
21     def __init__(self,
22                 v_reset = -80.0,
23                 e_rev_I = -75.0,
24                 v_rest = -75.0,
25                 v_thresh = -53.0,
26                 g_leak = 20.0,
27                 tau_syn_E= 10.,
28                 tau_syn_I= 10.):
```

```

29         self.dic = {
30             'v_reset'      : v_reset,    # mV
31             'e_rev_I'      : e_rev_I,    # mV
32             'v_rest'       : v_rest,     # mV
33             'v_thresh'     : v_thresh,   # mV
34             'g_leak'       : g_leak,     # nS
35             'tau_syn_E'    : tau_syn_E,   # ms
36             'tau_syn_I'    : tau_syn_I   # ms
37         }
38
39 # Helper class to store stimulation parameters
40 class StimParameters:
41     def __init__(self,
42                 useHardware,                # Bool
43                 ioutBase,                  #
44                 ifallBase,                 #
45                 neuronParams,              #
46                 firing_rate_exc,          # 25 >
47                 firing_rate_inh,          # 25 >
48                 numExcInputs = 20,         # Int
49                 numInhInputs = -1,         # Int
50                 spikesRecordPath = 'spikes.dat', #
51                 statisticsRecordFolder = 'data/', #
52                 numRuns = 1,               # Int
53                 numNeurons = 2,            # Int
54                 offset = 0,                # Int
55                 w_excSW = 1.e-16,          #
56                 w_inhSW = -1,              #
57                 w_exc = 1.0,               #
58                 w_inh = -1,                #
59                 expDuration = 10000,       # Int
60                 usePlot = False,           # Bool
61                 ratioSupthreshSubthresh = 0.8, #

```

```

62         deltaTfactor = 1, #
           factor is multiplied with tau syn E
63         numCorrInputs = 1 #
           number of inputs in a correlated
           input package
64     ):
65     self.useHardware = useHardware
66     self.numExcInputs = numExcInputs
67     if numInhInputs == -1:
68         self.numInhInputs = numExcInputs/1 # !!!!
           nicht mehr im Verhealtnis 1/4
69     else:
70         self.numInhInputs = numInhInputs
71     self.spikesRecordPath = spikesRecordPath
72     self.statisticsRecordFolder =
           statisticsRecordFolder
73     self.numRuns = numRuns
74     if useHardware:
75         self.numNeurons = numNeurons
76     else:
77         self.numNeurons = 2
78     self.offset = offset
79     if useHardware:
80         if w_inh < 0:
81             self.w_exc = w_exc
82             self.w_inh = w_exc*0.25
83         else:
84             self.w_exc = w_exc
85             self.w_inh = w_inh
86     else:
87         if w_inhSW < 0:
88             self.w_exc = w_excSW
89             self.w_inh = w_excSW*0.25
90         else:
91             self.w_exc = w_excSW
92             self.w_inh = w_inhSW
93     self.expDuration = expDuration
94     self.ioutBase = ioutBase
95     self.ifallBase = ifallBase
96     self.neuronParams = neuronParams.dic
97     self.firing_rate_exc = firing_rate_exc
98     self.firing_rate_inh = firing_rate_inh #
           CHANGE BACK TO firing_rate_inh
99     self.usePlot = False
100    self.ratioSupthreshSubthresh =
           ratioSupthreshSubthresh
101    self.numCorrInputs = numCorrInputs
102    self.deltaT = neuronParams.dic["tau_syn_E"]*
           deltaTfactor
103

```

```

104 # Control and configure the experiment
105 class Stimulation:
106     def __init__(self,
107                 stimParameters):
108         self.stimParameters = stimParameters
109         self.neuronParams = stimParameters.
110             neuronParams
111         self.ratioSupthreshSubthresh = stimParameters.
112             ratioSupthreshSubthresh
113         self.poisson_rng_exc = numpy.random
114         self.poisson_rng_exc.seed(int(time.time()*1000))
115         self.poisson_rng_inh = numpy.random
116
117     # setup the experiment with given parameters,
118     # so that it is runnable.
119     # the setup can be changed,
120     # all information is stored in class attributes.
121     def setup(self, usescope = False, workstationName="
122         station412"):
123         if self.stimParameters.useHardware:
124             pynnHW.setup(timestep=0.1,
125                         debug=False,
126                         useScope=usescope,
127                         mappingOffset=self.
128                             stimParameters.offset,
129                         calibOutputPins=False,
130                         calibTauMem=False,
131                         calibSynDrivers=False,
132                         calibVthresh=False,
133                         loglevel=0,
134                         logfile="logfile",
135                         ratioSuperthreshSubthresh = self
136                             .ratioSupthreshSubthresh,
137                         workstationName=workstationName)
138         self.neuron = pynnHW.create(pynnHW.
139             IF_facets_hardware1,
140                                     self.neuronParams
141                                     ,
142                                     n= self.
143                                     stimParameters
144                                     .numNeurons)
145
146     # create empty hardware simulation spike
147     sources
148     self.i_exc = pynnHW.create(pynnHW.
149         SpikeSourceArray,
150                                 n=self.
151                                 stimParameters.
152                                 numExcInputs)
153
154     self.i_inh = pynnHW.create(pynnHW.
155         SpikeSourceArray,

```



```

140         n=self.
           stimParameters.
           numInhInputs)
141     else:
142         pynnSW.setup(timestep=0.1)
143         self.neuron = pynnSW.create(pynnSW.
           IF_facets_hardware1,
144                                     self.neuronParams
           ,
145                                     n= self.
           stimParameters
           .numNeurons)
146         # create empty software simulation spike
           sources
147         self.i_exc = pynnSW.create(pynnSW.
           SpikeSourceArray,
148                                     n=self.
           stimParameters.
           numExcInputs)
149         self.i_inh = pynnSW.create(pynnSW.
           SpikeSourceArray,
150                                     n=self.
           stimParameters.
           numInhInputs)
151         # fill up with poisson spike trains
152         # the inputs are divided into packages,
153         # in which every spiketrain is the exact copy of
           the previous spiketrain,
154         # but with a delay of deltaT
155         count = 0
156         offset = self.stimParameters.deltaT
157         for e in self.i_exc:
158             if count%self.stimParameters.numCorrInputs ==
           0:
159                 # print "package nr " + str(count/self.
           stimParameters.numCorrInputs + 1)
160                 newSpikeTrain = poisson_gen.generate(
           start= 0.0,
161                                     duration= self.
           stimParameters.
           expDuration,
162                                     freq= self.
           stimParameters.
           firing_rate_exc,
163                                     rng= self.
           poisson_rng_exc
           )
164         else:
165             newSpikeTrain = numpy.array(newSpikeTrain
           )

```

```

166         newSpikeTrain += offset
167         for ii in range(len(newSpikeTrain)):
168             if newSpikeTrain[ii] > self.stimParameters.expDuration:
169                 newSpikeTrain[ii] -= self.stimParameters.expDuration
170         newSpikeTrain.sort()
171         # print newSpikeTrain[-1]
172         e.set_parameters(spike_times= newSpikeTrain)
173         count += 1
174     for i in self.i_inh:
175         newSpikeTrain = poisson_gen.generate(start=
176             0.0,
177             duration= self.stimParameters.expDuration,
178             freq= self.stimParameters.firing_rate_inh,
179             rng= self.poisson_rng_inh)
180         i.set_parameters(spike_times= newSpikeTrain)
181     if self.stimParameters.useHardware:
182         # adjust drvifallBase
183         pynnHW.hardware.hwa.drvifall_base['exc'] =
184             self.stimParameters.ifallBase
185         # adjust drvioutFall
186         pynnHW.hardware.hwa.drviout_base['exc'] =
187             self.stimParameters.ioutBase
188         # adjust drvifallBase
189         pynnHW.hardware.hwa.drvifall_base['inh'] =
190             self.stimParameters.ifallBase
191         # adjust drvioutFall
192         pynnHW.hardware.hwa.drviout_base['inh'] =
193             self.stimParameters.ioutBase
194         pynnHW.connect(self.i_exc,
195             self.neuron,
196             weight= self.stimParameters.w_exc,
197             synapse_type='excitatory')
198         pynnHW.connect(self.i_inh,
199             self.neuron,
200             weight= self.stimParameters.w_inh,
201             synapse_type='inhibitory')
202         pynnHW.record(self.neuron, self.stimParameters.spikesRecordPath)

```

```

198         else:
199             pymSW.connect(self.i_exc,
200                           self.neuron,
201                           weight=self.stimParameters.
202                               w_exc,
203                               synapse_type='excitatory')
204             pymSW.connect(self.i_inh,
205                           self.neuron,
206                           weight=self.stimParameters.
207                               w_inh,
208                               synapse_type='inhibitory')
209             pymSW.record(self.neuron, self.
210                           stimParameters.spikesRecordPath)
211         if self.stimParameters.usePlot:
212             self.rplot = myrasterplot.Rasterplot(self.
213                           stimParameters.expDuration,
214                                                   self.
215                                                   neuron
216                                                   )
217
218     def resetFiringRates(self):
219         self.firingRates = []
220
221     def run(self):
222         for i in range(self.stimParameters.numRuns):
223             if self.stimParameters.useHardware:
224                 pymHW.run(self.stimParameters.
225                             expDuration, ratioSuperthreshSubthresh
226                             = self.ratioSupthreshSubthresh)
227                 pymHW.end()
228             else:
229                 pymSW.run(self.stimParameters.
230                             expDuration)
231                 pymSW.end()
232             self.firingRates.append(firerate.firerate(
233                 self.stimParameters.expDuration,
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

```

226         #print self.firingRates
227         if self.stimParameters.usePlot:
228             plot.plot(self.stimParameters.
229                        spikesRecordPath,
230                        self.rplot)
231     def statistics(self):
232         self.firingRates = numpy.array(self.firingRates)
233         fr = firerate.averageFirerate(self.firingRates)
234         dr = firerate.sdeviationFirerate(self.firingRates
235                                          , fr)
236         tr = firerate.totalAverage(fr)
237         td = firerate.totalDeviation(dr)
238         firerate.printToFile(self.stimParameters.
239                              statisticsRecordFolder
240                              + '_outBase_',
241                              str(self.stimParameters.
242                                 ioutBase)
243                              + '_fallBase_',
244                              str(self.stimParameters.
245                                 ifallBase)
246                              + '.dat',
247                              fr, dr)
248         self.tr = tr*1000
249         self.td = td*1000
250         return self.tr
251
252     def printStatistics(self):
253         print "Statistics_OFR:"
254         print repr(self.tr) + '_' + '+_' + repr(self.td)

```

Listing 2: iteration.py

```

1  # Script to test various background stimulations with
2  # different
3  # iout base and ifall base parameters.
4  # Compare with software simulation.
5  # It is build in a modular way,
6  # so that any component can be replaced easily.
7  # by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-
8  # heidelberg.de
9  # 26.01.2011
10
11
12 # for given parameters
13 # get the sign of a number
14 # returns 1 if positive
15 # returns 0 if 0

```

```
16 # returns -1 if negative
17 def sign(number):
18     if number < 0: return -1
19     if number > 0: return 1
20     return 0
21
22 # checks if a number is NOT in an intervall
23 def outOfBound(number, mi, ma):
24     return (mi > number or ma < number)
25
26
27 class FitValue:
28     def __init__(self,
29                 target, # the value to target
30                 minValue, # minimal value of the
31                        # variable
32                 maxValue, # maximal value of the
33                        # variable
34                 tolerance = 3., # difference to target
35                        # in percent
36                 mIterations=10 # max iterations
37                 # last Result =? # the last result of the
38                 # experiment
39                 ):
40         self.target = target
41         self.variable = (minValue + maxValue/2.)
42         self.tolerance = tolerance
43         self.mi = minValue
44         self.ma = maxValue
45         self.iterations = 0
46         self.mIterations = mIterations
47         # self.lastResult = lastResult
48         # NOTE: This last attribute is automatically
49         # created
50         # by the following function.
51
52         #####
53         # getNewVariable(result)
54         ####
55         # returns the new value of the variable,
56         # returns -1 if there is no better result to expect
57         # returns 0 if the target is out of range or reached
58         # max iterations
59
60     def getNewVariable(self, result):
61         self.iterations = self.iterations +1
62         targetAcquired = abs(result-self.target)/self.
63             target < self.tolerance/100.
64
65         # Oh-Happy-Day-Scenario
66         if targetAcquired:
67             self.lastResult = result
```

```

59         print
60         print "target_acquired"
61         print
62         return -1
63     if self.iterations > self.mIterations:
64         print "reached_max_iterations"
65         return 0
66     # check if there has already been a previous result
67     # if hasattr(self, 'lastResult'):
68     #     check if (result-target) has same sign as (
69     #         lastResult-target)
70     #     if not, we have to turn around and decrease
71     #         the stepwidth
72     #     if not(sign(result-self.target)== sign(self.
73     #         lastResult-self.target)):
74     #         self.stepwidth = self.stepwidth/2.
75     #         print "decreasing stepwidth to " + str(
76     #             self.stepwidth)
77     # the result is smaller than the target
78     if result < self.target:
79         self.mi = self.variable
80         self.variable = (self.mi + self.ma)/2.
81     # the result is bigger than the target
82     else:
83         self.ma = self.variable
84         self.variable = (self.mi + self.ma)/2.
85     # self.lastResult = result
86     print
87     print "Step_#" + str(self.iterations)
88     print "New_Variable_=" + str(self.variable)
89     return self.variable
90
91     #####
92     # getStepwidth()
93     ###
94     # Returns the current stepwidth of the iteration.
95     # The return value can by interpreted as a max error
96     # of the current result.
97     def getStepwidth(self):
98         return self.ma - self.mi

```

Listing 3: firerate.py

```

1 # helper functions to get the firerate of neurons
2 # by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-
3   heidelberg.de
4 # 08.12.10
5 # review 22.12.10
6 # by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-
7   heidelberg.de
8 # review 11.01.11

```

```

7 # by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-
  heidelberg.de
8
9 # review 11.01.11
10 # changed from decimal to pylab
11 # import decimal
12 import pylab as p
13 import numpy as n
14
15 # calculate firerate for every neuron
16 # review 11.01.11
17 # load with pylab
18 def firerate(expDuration, numNeuron, dataPath):
19     try:
20         spikelist = p.loadtxt(dataPath)
21     except:
22         return n.array([0.]*numNeuron)
23     firelist = []
24     #print spikelist
25     for i in range(1, numNeuron+1):
26         spikes = spikelist[spikelist[:,1]==i]
27         firelist.append(float(len(spikes))/expDuration)
28     #print firelist
29     return n.array(firelist)
30
31 # review 11.01.11
32 # adapted to numpy array
33 def averageFirerate(firingRates):
34     numRuns = len(firingRates)
35     #print firingRates
36     #print type(firingRates)
37     fireList = n.mean(firingRates, axis=0)
38     #print fireList
39     return fireList
40
41 # review 11.01.11
42 # adapted to numpy array
43 def sdeviationFirerate(firingRates, avFiringRates):
44     devList = n.std(firingRates, axis=0)
45     #print devList
46     return devList
47
48 def printStatistics(avFiringRates, devList):
49     print 'Nr_Average_Firerate_____Standard_Deviation'
50     for i in range(len(avFiringRates)):
51         print repr(i).rjust(2), repr(avFiringRates[i]).
52             ljust(35), repr(devList[i]).ljust(35)
53
54 def printToFile(fileName, firingRates, devList):
55     f = open(fileName, 'w')

```

```

55     for i in range(len(firingRates)):
56         print >>f, repr(i).rjust(2), repr(firingRates[i])
           .ljust(20), repr(devList[i]).ljust(22)
57     f.close()
58
59 # 22.12.2010 total average firingrate calculation
60 def totalAverage(firerate):
61     tr = sum(firerate)/len(firerate)
62     return tr
63
64 # 17.01.2012 total deviation of firing rate
65 def totalDeviation(deviation):
66     s =0.0
67     for i in deviation:
68         s= s+i*i
69     return (s/len(deviation))**0.5

```

Listing 4: outFallexperiment.py

```

1 # Class to find the value of ioutBase with given
   ifallBase ,
2 # to match OFR with software simulation
3 # by Ioannis Kokkinos, ioannis.kokkinos@kip.uni-
   heidelberg.de
4 # 26.01.2011
5
6 import background as ba
7 import iteration as it
8
9 class Experiment:
10     def __init__(self ,
11                 ifallBase ,           # constant
12                 inputs ,             # number of exc inputs
13                 numRuns ,            #
14                 numNeurons = 192, #
15                 expDuration=6000, # CHANGE BACK TO 6000
16                 usePlot = False , #
17                 ratioSupthreshSubthres = 0.8 ,
18                 deltaTfactor = 1. ,
19                 numCorrInputs = 1
20                 ):
21         self.ifallBase = ifallBase
22         self.inputs     = inputs
23         self.numRuns    = numRuns
24         self.numNeurons = numNeurons
25         self.expDuration= expDuration
26         self.usePlot    = usePlot
27         self.ratioSupthreshSubthres =
           ratioSupthreshSubthres

```



```

28  #      self.refOFR          # will be created by
      the SW ref exp
29  #      self.refIFR          # will be created by
      the SW ref exp
30      self.w_excSW      = 0.00218
31      self.mi           = 0.0
32      self.ma           = 1.6
33      self.tolerance    = 1.1
34      self.mIt          = 8
35      self.deltaTfactor = deltaTfactor
36      self.numCorrInputs = numCorrInputs
37
38  # Start a software reference experiment with NEST
39  # the resulting input firing rate should produce the
      desired
40  # OFR (output firing rate)
41  # return a list with ifr, ofr and iterations
42  def refExp(self, tfr):
43      neuPar = ba.NeuronParams()
44      fv = it.FitValue(tfr,          # target
45                      1.,          # min
46                      25.,         # max
47                      tolerance = 1., # tolerance
48                      mIterations = 40 # max
49                      iterations
50                      )
51      result = 1
52      var = fv.variable
53      while(var > 0):
54          self.refIFR = var
55          stiPar = ba.StimParameters(False,
56                                     1.,
57                                     1.,
58                                     neuPar,
59                                     var,
60                                     var,
61                                     numExcInputs = self.
62                                     inputs,
63                                     numRuns = 1,
64                                     numNeurons = 2,
65                                     w_excSW = self.w_excSW
66                                     ,
67                                     expDuration = self.
68                                     expDuration,
69                                     usePlot = self.usePlot
70                                     ,
71                                     deltaTfactor = self.
72                                     deltaTfactor,
73                                     numCorrInputs = self.
74                                     numCorrInputs

```

```

68         )
69         stim = ba.Stimulation(stiPar)
70         stim.resetFiringRates()
71         if fv.iterations < 4:
72             for i in range(3):
73                 stim.setup()
74                 stim.run()
75         else:
76             for i in range(self.numRuns):
77                 stim.setup()
78                 stim.run()
79             result = stim.statistics()
80             stim.printStatistics()
81             var = fv.getNewVariable(result)
82         self.refOFR = result
83         return [self.refIFR, self.refOFR, fv.iterations]
84
85     # Start a software reference experiment with NEST
86     # the resulting output firing rate should be
87     # reproduceable by hardware
88     # return a list with ifr, ofr and iterations
89     def refDeltaT(self, deltaT):
90         self.deltaTfactor = deltaT
91         neuPar = ba.NeuronParams()
92         result = 1
93         stiPar = ba.StimParameters(False,
94                                     1.,
95                                     1.,
96                                     neuPar,
97                                     self.refIFR,
98                                     self.refIFR,
99                                     numExcInputs = self.
100                                     inputs,
101                                     numRuns = 1,
102                                     numNeurons = 2,
103                                     w_excSW = self.w_excSW
104                                     ,
105                                     expDuration = self.
106                                     expDuration,
107                                     usePlot = self.usePlot
108                                     ,
109                                     deltaTfactor = self.
110                                     deltaTfactor,
111                                     numCorrInputs = self.
112                                     numCorrInputs
113                                     )
114         stim = ba.Stimulation(stiPar)
115         stim.resetFiringRates()
116         for i in range(self.numRuns):
117             stim.setup()

```

```

112         stim.run()
113         result = stim.statistics()
114         stim.printStatistics()
115         self.refOFR = result
116         return result
117
118
119
120     # Start a experiment with hardware
121     # the resulting ioutBase should produce the TFR (target
122         firing rate)
123     def experiment(self, refOFR):
124         neuPar = ba.NeuronParams()
125         fv = it.FitValue(refOFR, #
126             target
127             self.mi, #
128                 min
129                 self.ma, #
130                 max
131                 tolerance = self.tolerance, #
132                 tolerance
133                 mIterations = self.mIt #
134                 max iterations
135         )
136     result = 1.
137     std = 0.
138     var = fv.variable
139     while(var > 0):
140         self.ioutBase = var
141         stiPar = ba.StimParameters(True,
142             var,
143             self.ifallBase,
144             neuPar,
145             self.refIFR,
146             self.refIFR,
147             numExcInputs = self.
148                 inputs,
149                 numRuns = 1,
150                 numNeurons = self.
151                     numNeurons,
152                 expDuration = self.
153                     expDuration,
154                 usePlot = self.usePlot
155             ,
156             deltaTfactor = self.
157                 deltaTfactor,
158             numCorrInputs = self.
159                 numCorrInputs
160         )
161     stim = ba.Stimulation(stiPar)

```

```
150         stim.resetFiringRates()
151         if fv.iterations < 3:
152             for i in range(3):
153                 stim.setup()
154                 stim.run()
155         else:
156             for i in range(self.numRuns):
157                 stim.setup()
158                 stim.run()
159             result = stim.statistics()
160             std = stim.td
161             err = fv.getStepwidth()
162             var = fv.getNewVariable(result)
163             stim.printStatistics()
164         return [result, std, fv.iterations, err]
165
166     def getIoutBase(self):
167         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 wissenschaftliches 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.