

NEST Topology Module: A Case-Based Tutorial

Author: Hans Ekkehard Plesser
Institution: Norwegian University of Life Sciences, Simula Research Laboratory,
RIKEN Brain Sciences Institute
Version: 0.2
Date: 13 August 2009
Copyright: The NEST Initiative (2009)
License: Creative Commons Attribution License

NOTE: The network generated by this script does generates dynamics in which the activity of the entire system, especially R_p and V_p oscillates with approx 5 Hz. This is different from the full model. Deviations are due to the different model type and the elimination of a number of connections, with no changes to the weights.

Introduction

This tutorial shows you how to implement a simplified version of the Hill-Tononi model of the early visual pathway using the NEST Topology module. The model is described in the paper

S. L. Hill and G. Tononi. Modeling Sleep and Wakefulness in the Thalamocortical System. *J Neurophysiology* **93**:1671-1698 (2005). Freely available via [doi 10.1152/jn.00915.2004](https://doi.org/10.1152/jn.00915.2004).

We simplify the model somewhat both to keep this tutorial a bit shorter, and because some details of the Hill-Tononi model are not currently supported by NEST. Simplifications include:

1. We use the `iaf_cond_alpha` neuron model, which is simpler than the Hill-Tononi model.
2. As the `iaf_cond_alpha` neuron model only supports two synapses (labeled “ex” and “in”), we only include AMPA and GABA_A synapses.
3. We ignore the secondary pathway (Ts, Rs, Vs), since it adds just more of the same from a technical point of view.
4. Synaptic delays follow a Gaussian distribution in the HT model. This implies actually a Gaussian distributions clipped at some small, non-zero delay, since delays must be positive. Currently, there is a bug in the Topology module when using clipped Gaussian distribution. We therefore draw delays from a uniform distribution.
5. Some further adaptations are given at the appropriate locations in the script.

This tutorial is divided in the following sections:

Philosophy Discusses the philosophy applied to model implementation in this tutorial

Preparations Necessary steps to use NEST and the Topology Module

Configurable Parameters Define adjustable network parameters

Neuron Models Define the neuron models needed by the network model

Populations Create Populations

Synapse models Define the synapse models used in the network model

Connections Create Connections

Example simulation Perform a small simulation for illustration. This section also discusses the setup for recording.

Philosophy

A network models has two essential components: *populations* and *projections*. We first use NEST's `CopyModel()` mechanism to create specific models for all populations and subpopulations in the network, and then create the populations using the Topology modules `CreateLayer()` function.

We use a two-stage process to create the connections, mainly because the same configurations are required for a number of projections: we first define dictionaries specifying the connections, then apply these dictionaries later.

The way in which we declare the network model here is an example. You should not consider it the last word: we expect to see a significant development in strategies and tools for network descriptions in the future. The following contributions to CNS*09 seem particularly interesting

- Ralf Ansorg & Lars Schwabe. Declarative model description and code generation for hybrid individual- and population-based simulations of the early visual system (P57);
- Sharon Crook, R. Angus Silver, & Pdraig Gleeson. Describing and exchanging models of neurons and neuronal networks with NeuroML (F1);

as well as the following paper which will apply in PLoS Computational Biology shortly:

- Eilen Nordlie, Marc-Oliver Gewaltig, & Hans Ekehard Plesser. Towards reproducible descriptions of neuronal network models.

Preparations

Ensure that we load NEST modules from the right place. You will not need this if you installed NEST in the standard location `/usr/local` or if your `PYTHONPATH` is set correctly.

```
125 import sys
126 sys.path = ["/Users/plesser/NEST/code/trunk/ins/lib/python2.5/site-packages"
             ] + sys.path
```

Load pynest

```
130 import nest
```

Load NEST Topology module, as of nest 1.9.r8375

```
133 import nest.topology as topo
```

Make sure we start with a clean slate, even if we re-run the script in the same Python session.

```
137 nest.ResetKernel()
```

Import math, we need Pi

```
140 import math
```

We want to plot below, too. We need to import `pylab` (not `matplotlib.pyplot`), since `pyreport` otherwise does not capture plot output.

```
145 import pylab
```

Configurable Parameters

Here we define those parameters that we take to be configurable. The choice of configurable parameters is obviously arbitrary, and in practice one would have far more configurable parameters. We restrict ourselves to:

- Network size in neurons `N`, each layer is `N x N`.
- Network size in subtended visual angle `visSize`, in degree.
- Temporal frequency of drifting grating input `f_dg`, in Hz.
- Spatial wavelength and direction of drifting grating input, `lambda_dg` and `phi_dg`, in degree/radian.
- Background firing rate of retinal nodes and modulation amplitude, `retDC` and `retAC`, in Hz.

- Simulation duration `simtime`; actual simulation is split into intervals of `sim_interval` length, so that the network state can be visualized in those intervals. Times are in ms.

```

174 Params = { 'N'           : 40,
175           'visSize'     : 8.0,
176           'f_dg'        : 2.0,
177           'lambda_dg'   : 2.0,
178           'phi_dg'       : 0.0,
179           'retDC'        : 30.0,
180           'retAC'        : 30.0,
181           'simtime'      : 100.0,
182           'sim_interval': 5.0
183         }

```

Neuron Models

We declare models in two steps:

1. We define a dictionary specifying the NEST neuron model to use as well as the parameters for that model.
2. We create three copies of this dictionary with parameters adjusted to the three model variants specified in Table~2 of Hill & TONI (2005) (cortical excitatory, cortical inhibitory, thalamic)

In addition, we declare the models for the stimulation and recording devices.

The general neuron model

We use the `iaf_cond_alpha` neuron, which is an integrate-and-fire neuron with two conductance-based synapses which have alpha-function time course. Any input with positive weights will automatically directed to the synapse labeled `_ex`, any with negative weights to the synapses labeled `_in`. We define **all** parameters explicitly here, so that no information is hidden in the model definition in NEST. `v_m` is the membrane potential to which the model neurons will be initialized. The model equations and parameters for the Hill-Tononi neuron model are given on pp. 1677f and Tables 2 and 3 in that paper. Note some peculiarities and adjustments:

- Hill & Tononi specify their model in terms of the membrane time constant, while the `iaf_cond_alpha` model is based on the membrane capacitance. Interestingly, conductances are unitless in the H&T model. We thus can use the time constant directly as membrane capacitance.
- The model includes sodium and potassium leak conductances. We combine these into a single one as follows:

$$-g_{NaL}(V - E_{Na}) - g_{KL}(V - E_K) = -(g_{NaL} + g_{KL}) \left(V - \frac{g_{NaL}E_{NaL} + g_{KL}E_K}{g_{NaL}g_{KL}} \right) \quad (1)$$

- We write the resulting expressions for `g_L` and `E_L` explicitly below, to avoid errors in copying from our pocket calculator.
- The paper gives a range of 1.0-1.85 for `g_{KL}`, we choose 1.5 here.
- The Hill-Tononi model has no explicit reset or refractory time. We arbitrarily set `V_reset` and `t_ref`.
- The paper uses double exponential time courses for the synaptic conductances, with separate time constants for the rising and falling flanks. Alpha functions have only a single time constant: we use twice the rising time constant given by Hill and Tononi.
- In the general model below, we use the values for the cortical excitatory cells as defaults. Values will then be adapted below.

```

243 nest.CopyModel('iaf_cond_alpha', 'NeuronModel',
244               params = { 'C_m'       : 16.0,
245                       'E_L'         : (0.2 * 30.0 + 1.5 * -90.0) / (0.2 +
246                       1.5),
247                       'g_L'         : 0.2 + 1.5,

```

```

247         'E_ex'      : 0.0,
248         'E_in'      : -70.0,
249         'V_reset'   : -60.0,
250         'V_th'      : -51.0,
251         't_ref'     : 2.0,
252         'tau_syn_ex' : 1.0,
253         'tau_syn_in' : 2.0,
254         'I_e'       : 0.0,
255         'V_m'       : -70.0})

```

Adaptation of models for different populations

We must copy the *NeuronModel* dictionary explicitly, otherwise Python would just create a reference.

Cortical excitatory cells

Parameters are the same as above, so we need not adapt anything

```

254 nest.CopyModel('NeuronModel', 'CtxExNeuron')

```

Cortical inhibitory cells

```

261 nest.CopyModel('NeuronModel', 'CtxInNeuron',
262               params = {'C_m' : 8.0,
263                       'V_th' : -53.0,
264                       't_ref' : 1.0})

```

Thalamic cells

```

269 nest.CopyModel('NeuronModel', 'ThalamicNeuron',
270               params = {'C_m' : 8.0,
271                       'V_th' : -53.0,
272                       't_ref' : 1.0,
273                       'E_in' : -80.0})

```

Input generating nodes

Input is generated by sinusoidally modulate Poisson generators, organized in a square layer of retina nodes. These nodes require a slightly more complicated initialization than all other elements of the network:

- Average firing rate DC, firing rate modulation depth AC, and temporal modulation frequency `Freq` are the same for all retinal nodes and are set directly below.
- The temporal phase `Phi` of each node depends on its position in the grating and can only be assigned after the retinal layer has been created. We therefore specify a function for initializing the phase `Phi`. This function will be called for each node.

```

294 def philnit(pos, lam, alpha):
295     '''Initializer function for phase of drifting grating nodes.
296
297     pos : position (x,y) of node, in degree
298     lam : wavelength of grating, in degree
299     alpha: angle of grating in radian, zero is horizontal
300
301     Returns number to be used as phase of AC Poisson generator.
302     '''
303     return 2.0 * math.pi / lam * (math.cos(alpha) * pos[0] + math.sin(alpha)
304           * pos[1])
305 nest.CopyModel('ac_poisson_generator', 'RetinaNode',
306               params = {'AC' : [Params['retAC']]},

```

```

307         'DC'      : Params[ 'retDC' ],
308         'Freq'    : [Params[ 'f_dg' ]],
309         'Phi'     : [0.0]})

```

Recording nodes

We use the new `multimeter` device for recording from the model neurons. At present, `iaf_cond_alpha` is one of few models supporting `multimeter` recording. Support for more models will be added soon; until then, you need to use `voltmeter` to record from other models.

We configure `multimeter` to record membrane potential to membrane potential at certain intervals to memory only. We record the GID of the recorded neurons, but not the time.

```

322 nest.CopyModel( 'multimeter', 'RecordingNode',
323               params = { 'interval' : Params[ 'sim_interval' ],
324                         'record_from' : [ 'V_m' ],
325                         'record_to' : [ 'memory' ],
326                         'withgid' : True,
327                         'withpath' : False,
328                         'withtime' : False })

```

Populations

We now create the neuron populations in the model, again in the form of Python dictionaries. We define them in order from eye via thalamus to cortex.

We first define a dictionary defining common properties for all populations

```

335 layerProps = { 'rows'      : Params[ 'N' ],
336               'columns'   : Params[ 'N' ],
337               'extent'    : [Params[ 'visSize' ], Params[ 'visSize' ]],
338               'edge_wrap': True}

```

This dictionary does not yet specify the elements to put into the layer, since they will differ from layer to layer. We will add them below by updating the `'elements'` dictionary entry for each population.

Retina

```

343 layerProps.update({ 'elements' : 'RetinaNode' })
344 retina = topo.CreateLayer(layerProps)

```

Now set phases of retinal oscillators; we use a list comprehension instead of a loop.

```

352 [nest.SetStatus([n], {"Phi": [philnit(topo.GetPosition([n]),
353                                     Params["lambda_dg"],
354                                     Params["phi_dg"])]})]
355 for n in nest.GetLeaves(retina)[0]

```

Thalamus

We first introduce specific neuron models for the thalamic relay cells and interneurons. These have identical properties, but by treating them as different models, we can address them specifically when building connections.

We use a list comprehension to do the model copies.

```

363 [nest.CopyModel('ThalamicNeuron', SpecificModel) for SpecificModel in (
    'TpRelay', 'TpInter')]

```

Now we can create the layer, with one relay cell and one interneuron per location:

```

366 layerProps.update({ 'elements' : [ 'TpRelay', 'TpInter' ] })
367 Tp = topo.CreateLayer(layerProps)

```

Reticular nucleus

We follow the same approach as above, even though we have only a single neuron in each location.

```

373 [nest.CopyModel('ThalamicNeuron', SpecificModel) for SpecificModel in ('
      RpNeuron',)]
374 layerProps.update({'elements': 'RpNeuron'})
375 Rp = topo.CreateLayer(layerProps)

```

Primary visual cortex

We follow again the same approach. We differentiate neuron types between layers and between pyramidal cells and interneurons. At each location, there are two pyramidal cells and one interneuron in each of layers 2-3, 4, and 5-6. Finally, we need to differentiate between vertically and horizontally tuned populations. When creating the populations, we create the vertically and the horizontally tuned populations as separate populations.

We use list comprehensions to create all neuron types:

```

389 [nest.CopyModel('CtxExNeuron', layer+'pyr') for layer in ('L23', 'L4', 'L56')]
390 [nest.CopyModel('CtxInNeuron', layer+'in') for layer in ('L23', 'L4', 'L56')]

```

Now we can create the populations, suffixes h and v indicate tuning

```

395 layerProps.update({'elements': [['L23pyr', 2, 'L23in', 1],
396                                ['L4pyr', 2, 'L4in', 1],
397                                ['L56pyr', 2, 'L56in', 1]]})
398 Vp_h = topo.CreateLayer(layerProps)
399 Vp_v = topo.CreateLayer(layerProps)

```

Collect all populations

For reference purposes, e.g., printing, we collect all populations in a tuple:

```

405 populations = (retina, Tp, Rp, Vp_h, Vp_v)

```

Inspection

We can now look at the network using *PrintNetwork*:

```

411 nest.PrintNetwork()

```

We can also try to plot a single layer in a network. For simplicity, we use Rp, which has only a single neuron per position.

```

414 Rppos = zip(*[topo.GetPosition([n]) for n in nest.GetLeaves(Rp)[0]])

```

The line above works as follows:

1. `GetLeaves` extracts all neurons from the *Rp* layer
2. For each neuron *n*, `GetPosition` obtains the position within the layer, in degrees.
3. The list comprehension `[... for n in ...]` results in a list of x-y-coordinate pairs. `zip(*[...])` turns this into one list of x- and one list of y-coordinates, as needed by `plot`.

```

423 pylab.plot(Rppos[0], Rppos[1], 'o')
424 axsz = Params['visSize']/2 + 0.2
425 pylab.axis([-axsz, axsz, -axsz, axsz])
426 pylab.title('Layer Rp')
427 pylab.show()

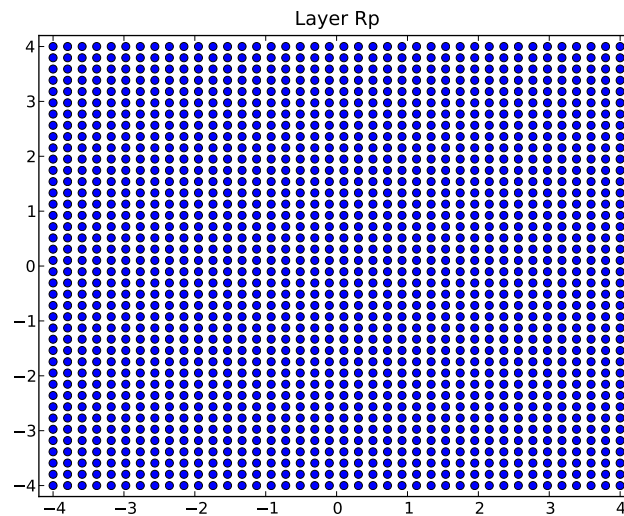
```

Synapse models

Actual synapse dynamics, e.g., properties such as the synaptic time course, time constants, reversal potentials, are properties of neuron models in NEST and we set them in section [Neuron models](#) above. When we refer to *synapse models* in NEST, we actually mean connectors which store information about connection weights and delays, as well as port numbers at the target neuron (`rport`) and implement synaptic plasticity. The latter two aspects are not relevant here.

We just use NEST's `static_synapse` connector but copy it to synapse models `AMPA` and `GABA_A` for the sake of explicitness. Weights and delays are set as needed in section [Connections](#) below, as they are different from projection to projection. De facto, the sign of the synaptic weight decides whether input via a connection is handle by the `_ex` or the `_in` synapse.

```
449 nest.CopyModel('static_synapse', 'AMPA')
```



```
451 nest.CopyModel('static_synapse', 'GABA_A')
```

Connections

Building connections is the most complex part of network construction. Connections are specified in Table 1 in the Hill-Tononi paper. As pointed out above, we only consider `AMPA` and `GABA_A` synapses here. Adding other synapses is tedious work, but should pose no new principal challenges. We also use a uniform in stead of a Gaussian distribution for the weights.

The model has two identical primary visual cortex populations, `Vp_v` and `Vp_h`, tuned to vertical and horizontal gratings, respectively. The *only* difference in the connection patterns between the two populations is the thalamocortical input to layers L4 and L5-6 is from a population of 8x2 and 2x8 grid locations, respectively. Furthermore, inhibitory connection in cortex go to the opposing orientation population as to the own.

To save us a lot of code doubling, we thus defined properties dictionaries for all connections first and then use this to connect both populations. We follow the subdivision of connections as in the Hill & Tononi paper.

Note: Hill & Tononi state that their model spans 8 degrees of visual angle and stimuli are specified according to this. On the other hand, all connection patterns are defined in terms of cell grid positions. Since the NEST Topology Module defines connection patterns in terms of the extent given in degrees, we need to apply the following scaling factor to all lengths in connections:

```
482 dpc = Params['visSize'] / (Params['N'] - 1)
```

We will collect all same-orientation cortico-cortical connections in

```
484 ccConnections = []
```

the cross-orientation cortico-cortical connections in

```
486 ccxConnections = []
```

and all cortico-thalamic connections in

```
489 ctConnections = []
```

Horizontal intralaminar

Note: “Horizontal” means “within the same cortical layer” in this case.

We first define a dictionary with the (most) common properties for horizontal intralaminar connection. We then create copies in which we adapt those values that need adapting, and

```

504 horIntraBase = {"connection_type": "divergent",
505               "synapse_model": "AMPA",
506               "mask": {"circular": {"radius": 12.0 * dpc}},
507               "kernel": {"gaussian": {"p_center": 0.05, "sigma": 7.5 * dpc
508                                }},
509               "weights": 1.0,
510               "delays": {"uniform": {"min": 1.75, "max": 2.25}}}
```

We use a loop to do the for for us. The loop runs over a list of dictionaries with all values that need updating

```

518 for conn in [{"sources": {"model": "L23pyr"}, "targets": {"model": "L23pyr"
519               }},
520             {"sources": {"model": "L23pyr"}, "targets": {"model": "L23in"
521               }},
522             {"sources": {"model": "L4pyr"}, "targets": {"model": "L4pyr"
523               },
524               "mask": {"circular": {"radius": 7.0 * dpc}}},
525             {"sources": {"model": "L4pyr"}, "targets": {"model": "L4in"
526               }},
527             {"sources": {"model": "L4pyr"}, "targets": {"model": "L4in"
528               },
529               "mask": {"circular": {"radius": 7.0 * dpc}}},
530             {"sources": {"model": "L56pyr"}, "targets": {"model": "L56pyr"
531               }},
532             {"sources": {"model": "L56pyr"}, "targets": {"model": "L56in"
533               }},
534             {"sources": {"model": "L56pyr"}, "targets": {"model": "L56in"
535               }}]:
536     ndict = horIntraBase.copy()
537     ndict.update(conn)
538     ccConnections.append(ndict)
```

Vertical intralaminar

Note: "Vertical" means "between cortical layers" in this case.

We proceed as above.

```

530 verIntraBase = {"connection_type": "divergent",
531               "synapse_model": "AMPA",
532               "mask": {"circular": {"radius": 2.0 * dpc}},
533               "kernel": {"gaussian": {"p_center": 1.0, "sigma": 7.5 * dpc
534                                }},
535               "weights": 2.0,
536               "delays": {"uniform": {"min": 1.75, "max": 2.25}}}
```

```

537 for conn in [{"sources": {"model": "L23pyr"}, "targets": {"model": "L56pyr"
538               }, "weights": 1.0},
539             {"sources": {"model": "L23pyr"}, "targets": {"model": "L23in"
540               }, "weights": 1.0},
541             {"sources": {"model": "L4pyr"}, "targets": {"model": "L23pyr"
542               }},
543             {"sources": {"model": "L4pyr"}, "targets": {"model": "L23in"
544               }},
545             {"sources": {"model": "L56pyr"}, "targets": {"model": "L23pyr"
546               }},
547             {"sources": {"model": "L56pyr"}, "targets": {"model": "L23in"
548               }},
549             {"sources": {"model": "L56pyr"}, "targets": {"model": "L4pyr"
550               }},
551             {"sources": {"model": "L56pyr"}, "targets": {"model": "L4in"
552               }}]:
553     ndict = verIntraBase.copy()
554     ndict.update(conn)
555     ccConnections.append(ndict)
```

Intracortical inhibitory

We proceed as above, with the following difference: each connection is added to the same-orientation and the cross-orientation list of connections.

Note: Weights increased from -1.0 to -2.0, to make up for missing GabaB

Note that we have to specify the **weight with negative sign** to make the connections inhibitory.

```

561 intralnhBase = {"connection_type": "divergent",
562               "synapse_model": "GABA_A",
563               "mask": {"circular": {"radius": 7.0 * dpc}},
564               "kernel": {"gaussian": {"p_center": 0.25, "sigma": 7.5 * dpc
565                                   }},
566               "weights": -2.0,
567               "delays": {"uniform": {"min": 1.75, "max": 2.25}}}
```

We use a loop to do the for for us. The loop runs over a list of dictionaries with all values that need updating

```

574 for conn in [{"sources": {"model": "L23in"}, "targets": {"model": "L23pyr"
575               }},
576             {"sources": {"model": "L23in"}, "targets": {"model": "L23in"
577               }},
578             {"sources": {"model": "L4in"}, "targets": {"model": "L4pyr"
579               }},
580             {"sources": {"model": "L4in"}, "targets": {"model": "L4in"
581               }},
582             {"sources": {"model": "L56in"}, "targets": {"model": "L56pyr"
583               }},
584             {"sources": {"model": "L56in"}, "targets": {"model": "L56in" }}
585         ]:
586     ndict = intralnhBase.copy()
587     ndict.update(conn)
588     ccConnections.append(ndict)
589     ccxConnections.append(ndict)
```

Corticothalamic

```

583 corThalBase = {"connection_type": "divergent",
584               "synapse_model": "AMPA",
585               "mask": {"circular": {"radius": 5.0 * dpc}},
586               "kernel": {"gaussian": {"p_center": 0.5, "sigma": 7.5 * dpc
587                                   }},
588               "weights": 1.0,
589               "delays": {"uniform": {"min": 7.5, "max": 8.5}}}
```

We use a loop to do the for for us. The loop runs over a list of dictionaries with all values that need updating

```

591 for conn in [{"sources": {"model": "L56pyr"}, "targets": {"model": "TpRelay"
592               }},
593             {"sources": {"model": "L56pyr"}, "targets": {"model": "TpInter"
594               }}]:
595     ndict = intralnhBase.copy()
596     ndict.update(conn)
597     ctConnections.append(ndict)
```

Corticoreticular

In this case, there is only a single connection, so we write the dictionary itself; it is very similar to the corThalBase, and to show that, we copy first, then update. We need no **targets** entry, since Rp has only one neuron per location.

```

599 corRet = corThalBase.copy()
600 corRet.update({"sources": {"model": "L56pyr"}, "weights": 2.5})
```

Build all connections beginning in cortex

Cortico-cortical, same orientation

```
607 [topo.ConnectLayer(Vp_h, Vp_h, conn) for conn in ccConnections]
608 [topo.ConnectLayer(Vp_v, Vp_v, conn) for conn in ccConnections]
```

Cortico-cortical, cross-orientation

```
611 [topo.ConnectLayer(Vp_h, Vp_v, conn) for conn in ccxConnections]
612 [topo.ConnectLayer(Vp_v, Vp_h, conn) for conn in ccxConnections]
```

Cortico-thalamic connections

```
615 [topo.ConnectLayer(Vp_h, Tp, conn) for conn in ctConnections]
616 [topo.ConnectLayer(Vp_v, Tp, conn) for conn in ctConnections]
617 topo.ConnectLayer(Vp_h, Rp, corRet)
618 topo.ConnectLayer(Vp_v, Rp, corRet)
```

Thalamo-cortical connections

Note: According to the text on p. 1674, bottom right, of the Hill & Tononi paper, thalamocortical connections are created by selecting from the thalamic population for each L4 pyramidal cell, ie, are *convergent* connections.

We first handle the rectangular thalamocortical connections.

```
634 thalCorRect = {"connection_type": "convergent",
635               "sources": {"model": "TpRelay"},
636               "synapse_model": "AMPA",
637               "weights": 5.0,
638               "delays": {"uniform": {"min": 2.75, "max": 3.25}}}
```

Horizontally tuned

```
637 thalCorRect.update({"mask": {"rectangular": {"lower_left": [-4.0*dpc, -1.0*
638               dpc],
639               "upper_right": [ 4.0*dpc,  1.0*
640               dpc]}}})
639 for conn in [{"targets": {"model": "L4pyr"}, "kernel": 0.5},
640             {"targets": {"model": "L56pyr"}, "kernel": 0.3}]:
641     thalCorRect.update(conn)
642     topo.ConnectLayer(Tp, Vp_h, thalCorRect)
```

Vertically tuned

```
645 thalCorRect.update({"mask": {"rectangular": {"lower_left": [-1.0*dpc, -4.0*
646               dpc],
647               "upper_right": [ 1.0*dpc,  4.0*
648               dpc]}}})
649 for conn in [{"targets": {"model": "L4pyr"}, "kernel": 0.5},
650             {"targets": {"model": "L56pyr"}, "kernel": 0.3}]:
651     thalCorRect.update(conn)
652     topo.ConnectLayer(Tp, Vp_v, thalCorRect)
```

Diffuse connections

```
658 thalCorDiff = {"connection_type": "convergent",
659               "sources": {"model": "TpRelay"},
660               "synapse_model": "AMPA",
661               "weights": 5.0,
662               "mask": {"circular": {"radius": 5.0 * dpc}},
663               "kernel": {"gaussian": {"p_center": 0.1, "sigma": 7.5 * dpc
664               }},
665               "delays": {"uniform": {"min": 2.75, "max": 3.25}}
666 for conn in [{"targets": {"model": "L4pyr"}},
667             {"targets": {"model": "L56pyr"}}]:
668     thalCorDiff.update(conn)
669     topo.ConnectLayer(Tp, Vp_h, thalCorDiff)
```

```
670 topo.ConnectLayer(Tp, Vp_v, thalCorDiff)
```

Thalamic connections

Connections inside thalamus, including Rp

Note: In Hill & Tononi, the inhibition between Rp cells is mediated by GABA_B receptors. We use GABA_A receptors here to provide some self-dampening of Rp.

Note: The following code had a serious bug in v. 0.1: During the first iteration of the loop, “synapse_model” and “weights” were set to “AMPA” and “0.1”, respectively and remained unchanged, so that all connections were created as excitatory connections, even though they should have been inhibitory. We now specify synapse_model and weight explicitly for each connection to avoid this.

```
682 thalBase = {"connection_type": "divergent",
683            "delays": {"uniform": {"min": 1.75, "max": 2.25}}}
684
685
686 for src, tgt, conn in [(Tp, Rp, {"sources": {"model": "TpRelay"},
687                               "synapse_model": "AMPA",
688                               "mask": {"circular": {"radius": 2.0 * dpc
689                                                }},
689                               "kernel": {"gaussian": {"p_center": 1.0, "
690                                                sigma": 7.5 * dpc}}},
691                               "weights": 2.0}),
692                       (Tp, Tp, {"sources": {"model": "TpInter"},
693                               "targets": {"model": "TpRelay"},
694                               "synapse_model": "GABAA",
695                               "weights": -1.0,
696                               "mask": {"circular": {"radius": 2.0 * dpc
697                                                }},
698                               "kernel": {"gaussian": {"p_center": 0.25, "
699                                                sigma": 7.5 * dpc}}}),
700                       (Tp, Tp, {"sources": {"model": "TpInter"},
701                               "targets": {"model": "TpInter"},
702                               "synapse_model": "GABAA",
703                               "weights": -1.0,
704                               "mask": {"circular": {"radius": 2.0 * dpc
705                                                }},
706                               "kernel": {"gaussian": {"p_center": 0.25, "
707                                                sigma": 7.5 * dpc}}}),
708                       (Rp, Tp, {"targets": {"model": "TpRelay"},
709                               "synapse_model": "GABAA",
710                               "weights": -1.0,
711                               "mask": {"circular": {"radius": 12.0 * dpc
712                                                }},
713                               "kernel": {"gaussian": {"p_center": 0.15, "
714                                                sigma": 7.5 * dpc}}}),
715                       (Rp, Tp, {"targets": {"model": "TpInter"},
716                               "synapse_model": "GABAA",
717                               "weights": -1.0,
718                               "mask": {"circular": {"radius": 12.0 * dpc
719                                                }},
720                               "kernel": {"gaussian": {"p_center": 0.15, "
721                                                sigma": 7.5 * dpc}}}),
722                       (Rp, Rp, {"targets": {"model": "RpNeuron"},
723                               "synapse_model": "GABAA",
724                               "weights": -1.0,
725                               "mask": {"circular": {"radius": 12.0 * dpc
726                                                }},
727                               "kernel": {"gaussian": {"p_center": 0.5, "
728                                                sigma": 7.5 * dpc}}})]:
729     thalBase.update(conn)
```

```
719     topo.ConnectLayer(src, tgt, thalBase)
```

Thalamic input

Input to the thalamus from the retina.

Note: Hill & Tononi specify a delay of 0 ms for this connection. We use 1 ms here.

```
731 retThal = {"connection_type": "divergent",
732           "synapse_model": "AMPA",
733           "mask": {"circular": {"radius": 1.0 * dpc}},
734           "kernel": {"gaussian": {"p_center": 0.75, "sigma": 2.5 * dpc}},
735           "weights": 10.0,
736           "delays": 1.0}
737
738 for conn in [{"targets": {"model": "TpRelay"}},
739             {"targets": {"model": "TpInter"}}]:
740     retThal.update(conn)
741     topo.ConnectLayer(retina, Tp, retThal)
```

Checks on connections

As a very simple check on the connections created, we inspect the connections from the central node of various layers. We obtain all connections and plot the target neuron locations.

```
752
753 def connPlot(spop, smod, tmod, syn, titl):
754     '''Plot connections.
755         spop, smod  source population and model
756         tmod       target model
757         syn        synapse type
758         titl       figure title'''
759
760     # center location, cast to int since 'pyreport' makes all div double
761     ctr = [int(Params["N"]/2), int(Params["N"]/2)]
762
763     # get element at center
764     src = topo.GetElement(spop, ctr)
765
766     # if center element is not individual neuron, find one
767     # neuron of desired model in center element
768     if nest.GetStatus(src, 'model')[0] == 'subnet':
769         src = [n for n in nest.GetLeaves(src)[0]
770              if nest.GetStatus([n], 'model')[0]==smod][:1]
771
772     # get position (in degrees) of chosen source neuron
773     srcpos = topo.GetPosition(src)
774
775     # get all targets
776     tgts = nest.GetConnections(src, syn)[0]['targets']
777
778     # pick targets of correct model type, get their positions,
779     # convert list of (x,y) pairs to pair of x- and y-lists
780     pos = zip(*[topo.GetPosition([n])
781              for n in tgts if nest.GetStatus([n],
782              'model')[0]==tmod])
783
784     # plot source neuron in red, slightly larger, targets on blue
785     pylab.clf()
786     pylab.plot(pos[0], pos[1], 'bo', markersize=5, zorder=99, label='Targets
787                ')
788     pylab.plot(srcpos[:1], srcpos[1:], 'ro', markersize=10,
```

```

788         markeredgecolor='r', zorder=1, label='Source')
789     axsz = Params['visSize']/2 + 0.2
790     pylab.axis([-axsz, axsz, -axsz, axsz])
791     pylab.title(titl)
792     pylab.legend()
793
794     # pylab.show() required for 'pyreport'
795     pylab.show()

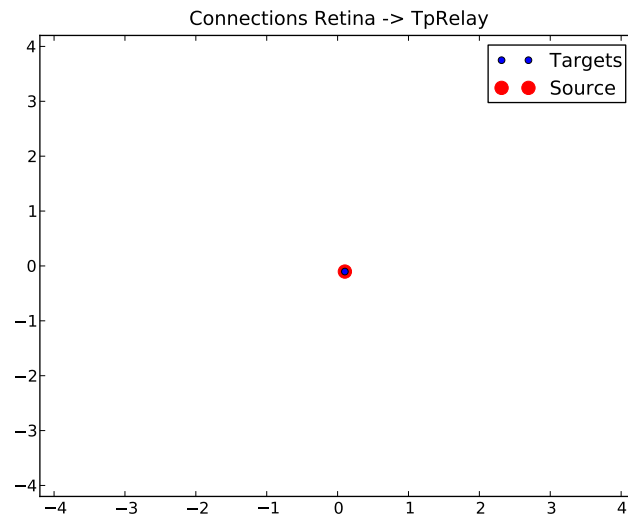
```

show Retina to TpRelay

```

792 connPlot(retina, '', 'TpRelay', 'AMPA', 'Connections Retina -> TpRelay')

```



show TpRelay to L4pyr

```

795 connPlot(Tp, 'TpRelay', 'L4pyr', 'AMPA', 'Connections TpRelay -> Vp L4pyr')

```

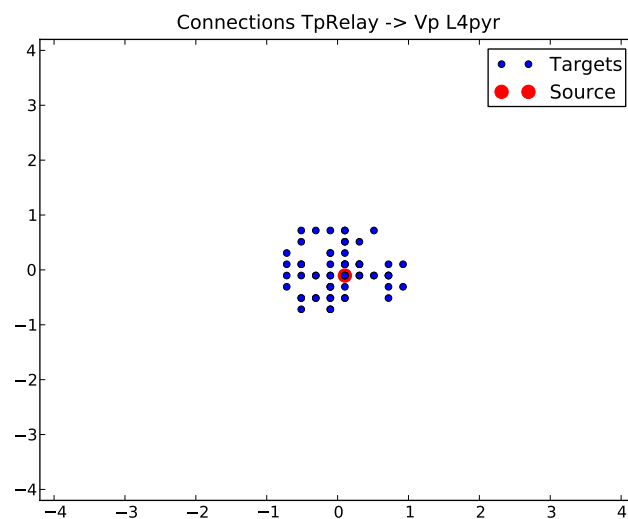
Recording devices

This recording device setup is a bit makeshift. For each population we want to record from, we create one `multimeter`, then select all nodes of the right model from the target population and connect. `loc` is the subplot location for the layer.

```

803 recorders = {}

```



```

812 for name, loc, population, model in [('TpRelay', 1, Tp, 'TpRelay'),

```

```

813         ('Rp'           , 2, Rp   , 'RpNeuron' ),
814         ('Vp_v L4pyr' , 3, Vp_v , 'L4pyr' ),
815         ('Vp_h L4pyr' , 4, Vp_h , 'L4pyr' )]:
816     recorders[name] = (nest.Create('RecordingNode'), loc)
817     tgts = [nd for nd in nest.GetLeaves(population)[0]
818            if nest.GetStatus([nd], 'model')[0]==model]
819     nest.DivergentConnect(recorders[name][0], tgts)

```

Example simulation

This simulation is set up to create a step-wise visualization of the membrane potential. To do so, we simulate `sim_interval` milliseconds at a time, then read out data from the multimeters, clear data from the multimeters and plot the data as pseudocolor plots.

show time during simulation

```

824 nest.SetStatus([0], {'print_time': True})

```

lower and upper limits for color scale, for each of the four populations recorded.

```

827 vmn=[-80, -80, -80, -80]
828 vmx=[-50, -50, -50, -50]
829
830 nest.Simulate(Params['sim_interval'])

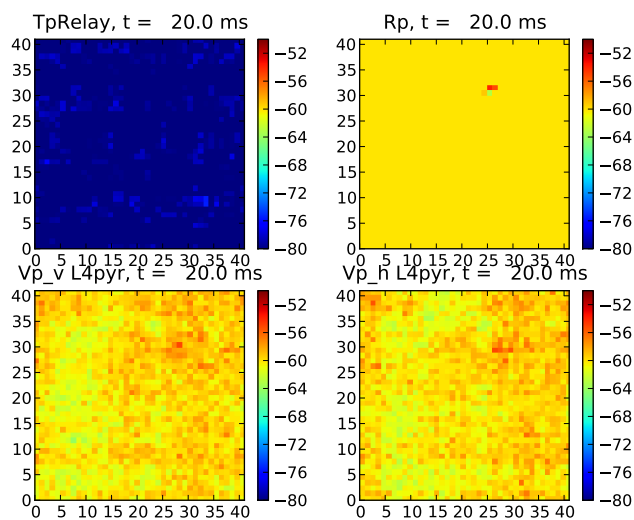
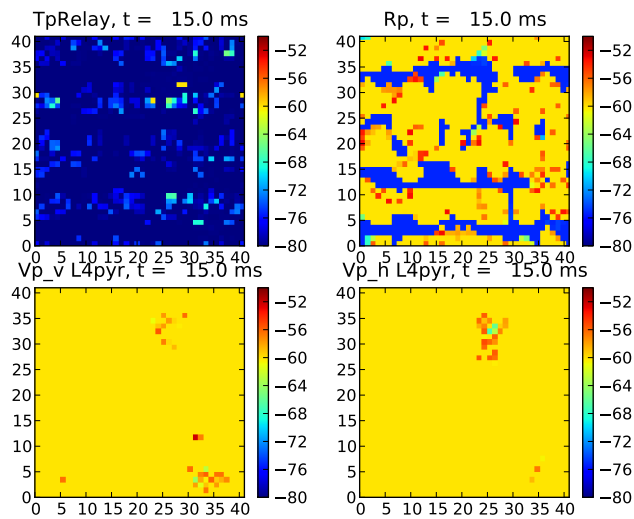
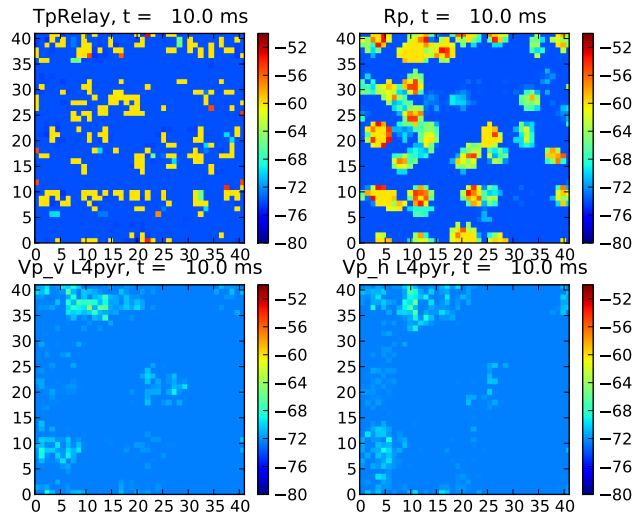
```

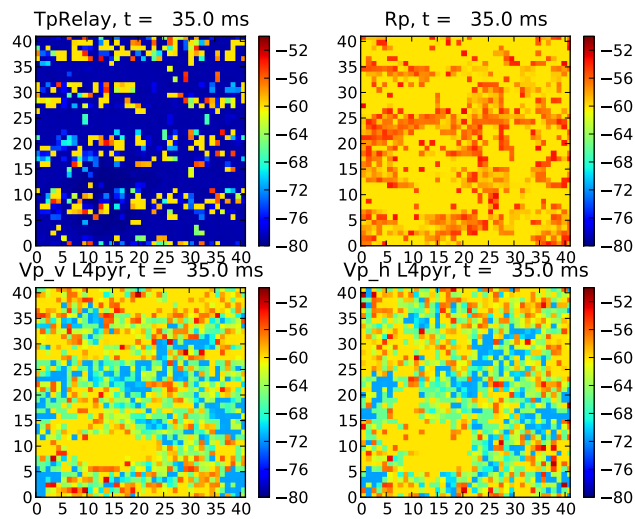
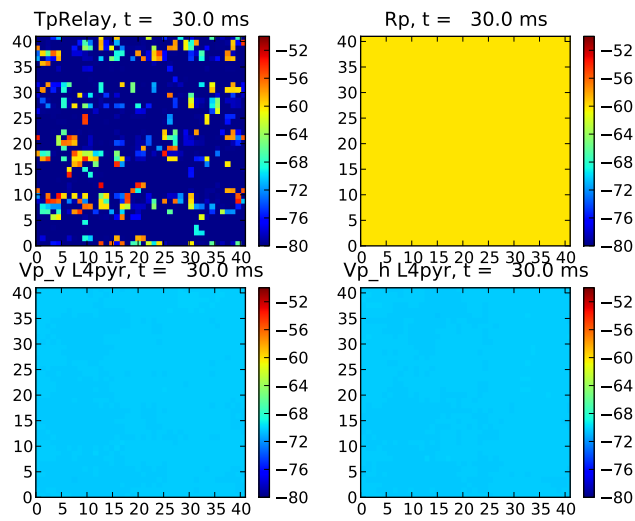
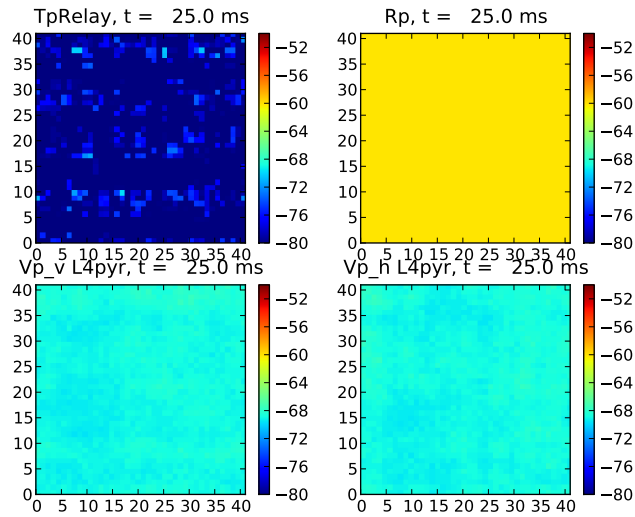
loop over simulation intervals

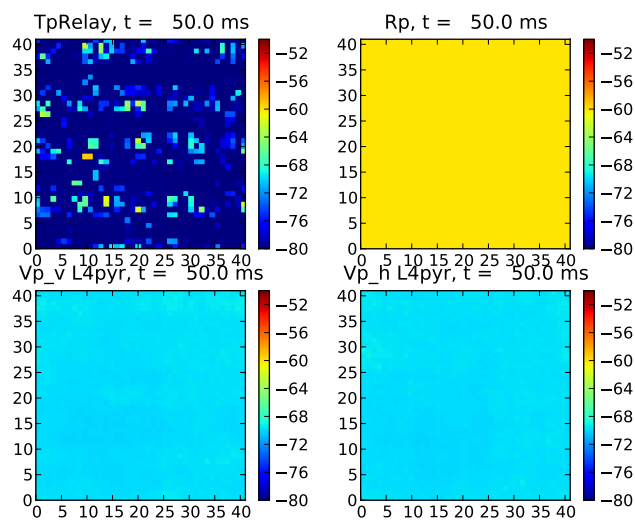
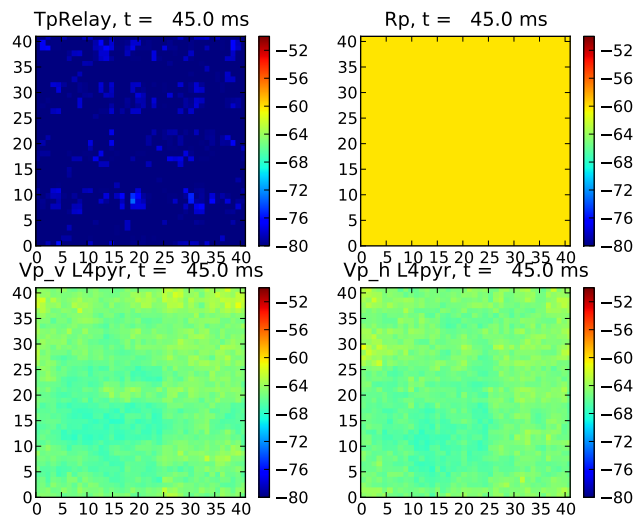
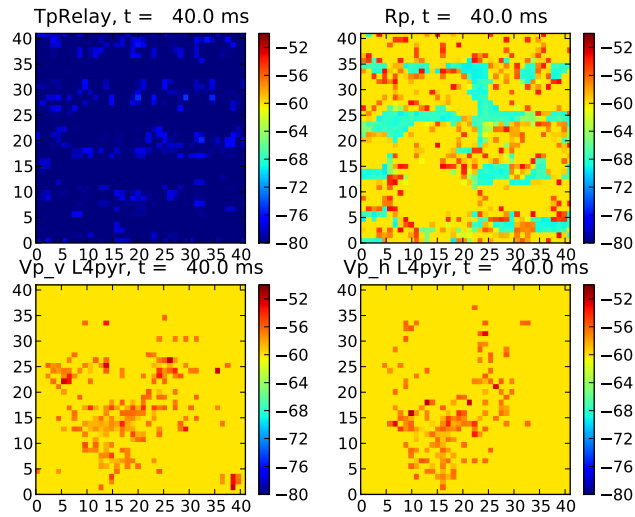
```

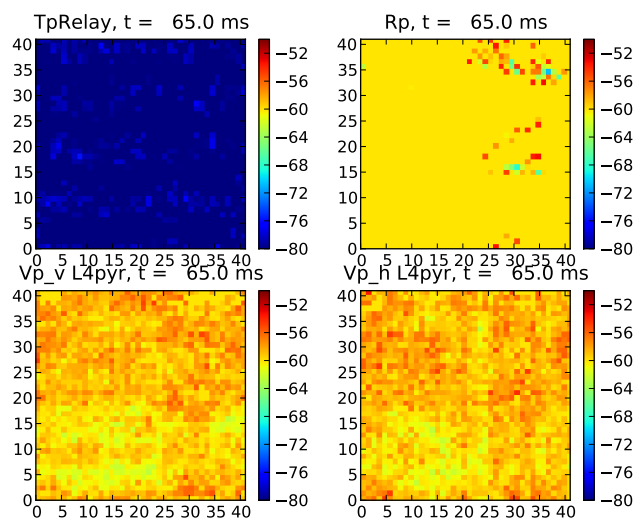
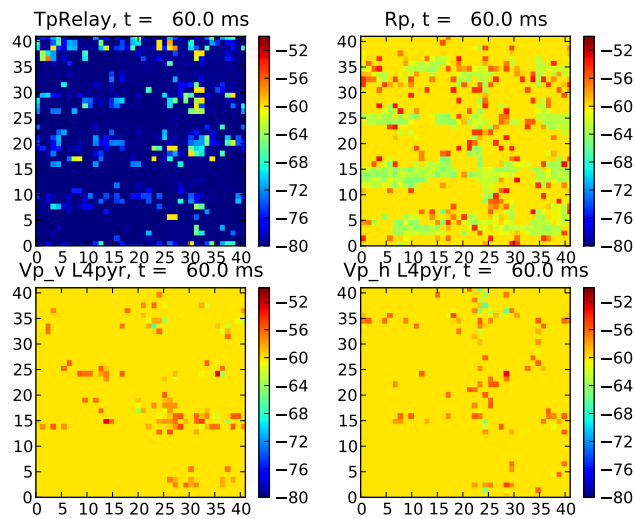
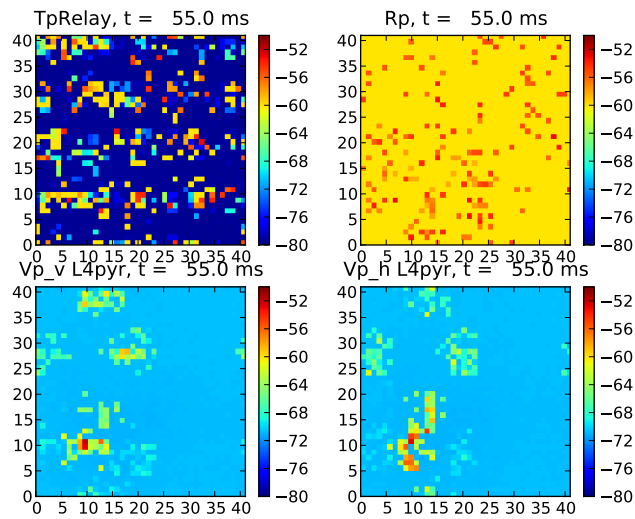
834 for t in pylab.arange(Params['sim_interval'], Params['simtime'], Params['
      sim_interval']):
835
836     # do the simulation
837     nest.Simulate(Params['sim_interval'])
838
839     # clear figure and choose colormap
840     pylab.clf()
841     pylab.jet()
842
843     # now plot data from each recorder in turn, assume four recorders
844     for name, r in recorders.iteritems():
845         rec = r[0]
846         sp = r[1]
847         pylab.subplot(2,2,sp)
848         d = nest.GetStatus(rec)[0]['events']['V_m']
849
850         if len(d) != Params['N']**2:
851             # cortical layer with two neurons in each location, take average
852             d = 0.5 * ( d[:, :2] + d[1: :2] )
853
854         # clear data from multimeter
855         nest.SetStatus(rec, {'n_events': 0})
856         pylab.imshow(pylab.reshape(d, (Params['N'], Params['N'])),
857                    aspect='equal', interpolation='nearest',
858                    extent=(0, Params['N']+1, 0, Params['N']+1),
859                    vmin=vmn[sp-1], vmax=vmx[sp-1])
860         pylab.colorbar()
861         pylab.title(name + ', t = %6.1f ms' % nest.GetKernelStatus()['time']
862                   )
863
864     # required by 'pyreport'
865     pylab.show()

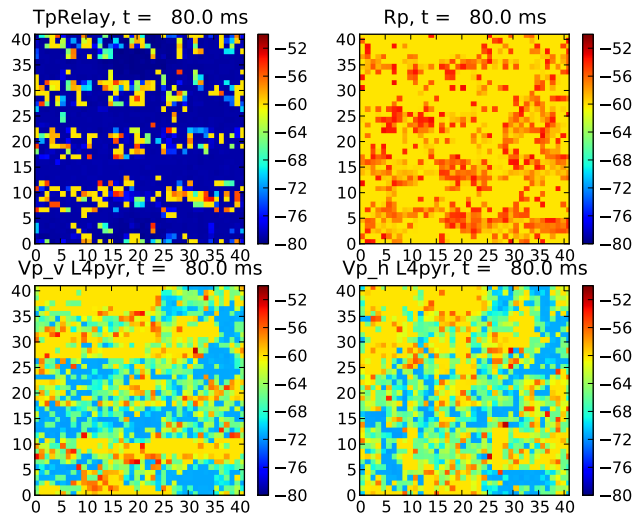
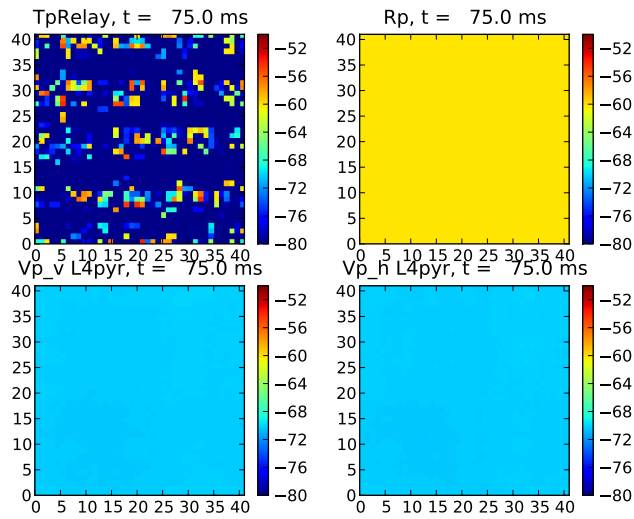
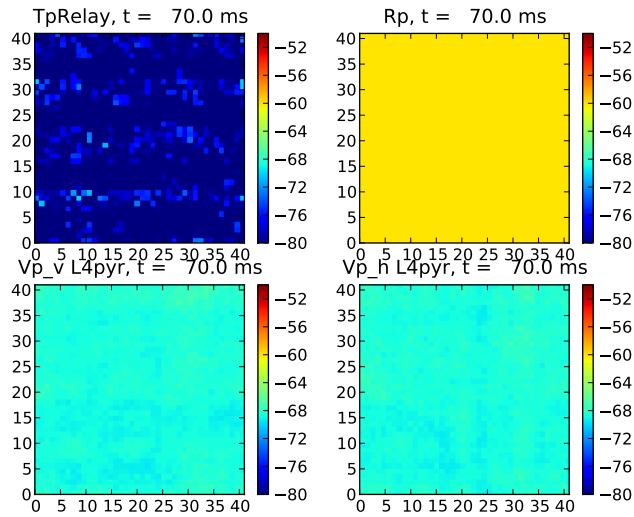
```

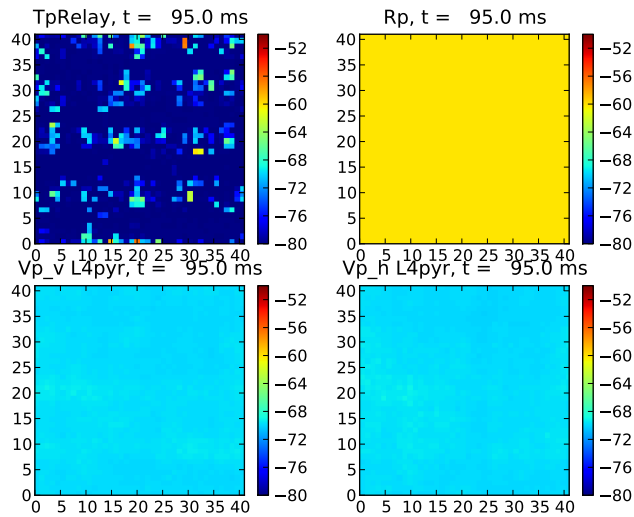
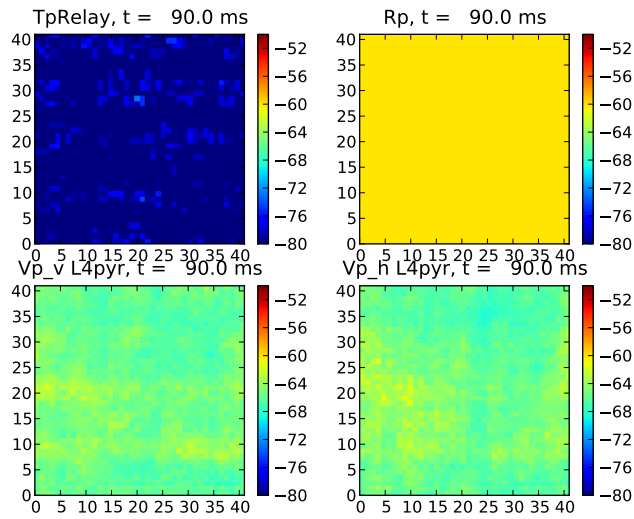
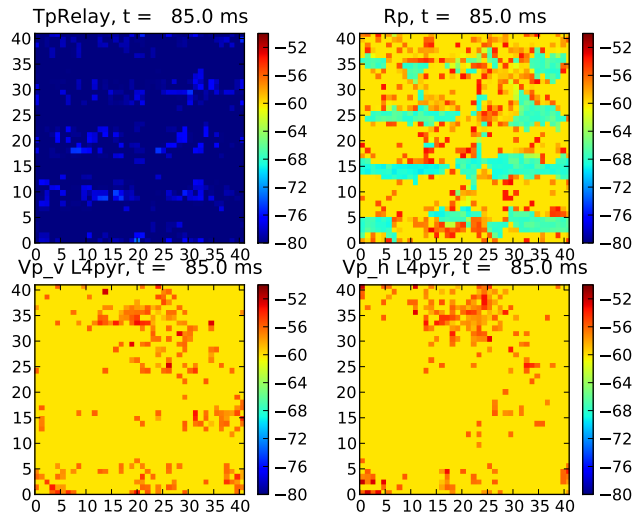


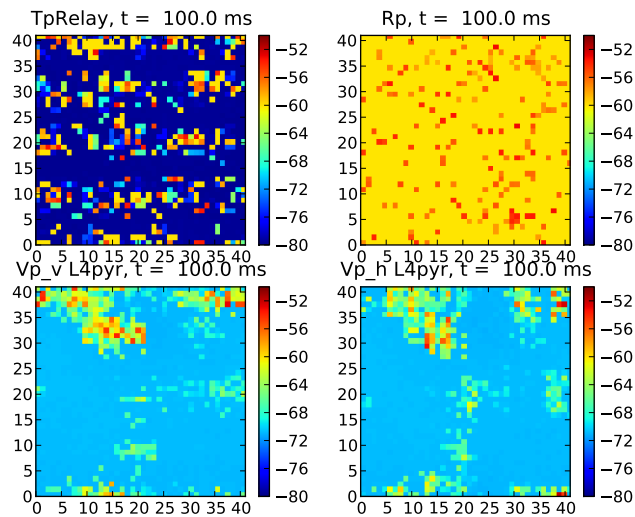












just for some information at the end

866

```
print nest.GetKernelStatus()
```

```
{'rng_seeds': [1], 'to_do': 0, 'rng_buffsize': 131072, 'data_path': '', '
  grng_seed': 2, 'local_num_threads': 1, 'off_grid_spiking': False, '
  network_size': 49610, 'print_time': True, 'dict_miss_is_error': True, '
  max_delay': 8.5, 'tics_per_step': 100, 'T_max': 26843545.5, '
  communicate_allgather': True, 'T_min': -26843545.5, 'ms_per_tic':
  0.001, 'num_processes': 1, 'overwrite_files': False, '
  total_num_virtual_procs': 1, 'tics_per_ms': 1000.0, 'data_prefix': '',
  'num_connections': 4015087, 'time': 100.0, 'resolution':
  0.10000000000000001, 'min_delay': 1.0}
```