NEST YAML PLATFORM (NYP)
v0.1
User manual (v0.2)

André Sevenius Nilsen

Brain Signalling Group
Department of Physiology
Institute of Basic Medical Sciences
University of Oslo

Please contact sevenius.nilsen@gmail.com if you want to use this platform or the accompanying “ht_model” in commercial or scientific work. Educational purposes are exempt.

1. Overview

1.1 Purpose

The purpose of this platform is to implement an easy to use, flexible, expandable, and compact human readable way to build models for simulation using the NEST platform, as well as offer a way to extract data in standard matrices for post-processing, and a selection of graphical outputs. The platform works around YML files, which are structured text files that contain all the parameters needed to build and simulate the a model in NEST. The platform is merely the interpreter of these files. YML files offer an two main improvements over that of using Python or PyNN to instruct the creating and simulation of networks in NEST. First, they are easier to read, create, change, and store, and secondly, they lower the threshold for starting with building and simulating networks in NEST. The current version (v0.1) only uses YAML files (Yet Another Markup Language) as the main way to configure the network and simulation, however this might be expanded upon in the future.

YAML files are in essence a way to write up lists, dictionaries, and tuples (if you're familiar with Python). More information can be found in section 3 (specifically 3.2).

The platform aims to provide a two step process of building networks: 1) populate the YML files with all the parameters you want, 2) press go.

The platform thus facilitates automatic output of simulation data in a more intuitive and flexible manner than that inherent in NEST, as well as automatic generation of premade plots from your simulation, to give a quick overview of your results.

In the future, the aim of this platform is to be more robust in terms of YML inputs, more flexible (opening up for more possible use cases), have a range of output formats and data types, have the option to manually control each step of the process and write custom code where needed, and create a series of plots that are useful to have after a simulation.

The current release is stable, and fully functional, albeit missing several optional features. It's also lacking robustness in terms of user errors when populating the YML files with parameters, as well as proper error handling and informative error information.

I hope this platform will be useful to you!

1.2 General instructions

The first thing you need to do is (in v0.1, expect this to change) to make a folder, call it whatever you want, and extract this package there. You should have the following files: auxillary_functions.py, checking_functions.py, io_functions.py, main.py, nest_interface.py, and output_functions.py, as well as two folders; example, and ht_model.

Then, after making sure you have all the dependencies required to run the platform (see appendix; dependencies), open a terminal window, move to the folder where you placed this package, and type the following:

```
python main.py "example"
```

And that's it. If you go to "example/output" you should now see some raster plots, some line plots, along with a "raw_data_output.mat" file. The latter file can be imported into MatLab or python or any other platform that can load Matlab files.

To create your very own network to simulate, the easiest is to simply copy the "example" folder and all it's content and give the copy a new name, for example "my_network". Then, go into the folders "my_network/network" and "my_network/simulation" and start changing parameters, with the help of this little user manual.

As of v0.1, the platform is rather unforgiving and lacking in output, but with some patience, trial and error, and double checking everything according to NEST and this pdf, you should be rolling in no time.

Good luck!

1.3 About the documentation

This documentation is split into two parts. The first is this part with some general comments, the second part is about the general functionality, what the model does, what commands you have available if you want manually control things a bit more (using Python), and a general primer on YAML syntax worded specifically for this document (i.e. terms may vary from official lingo).

1.3.1 Documentation syntax

Code (python, bash, YAML) is specified if it should be in a YAML file, from a *NIX shell (terminal), or from a python environment, and is placed in a code box like so:

```
#Here is a python comment

python_code = numpy.array[1,2,3]

python python_code.py

yaml_structure:
    yaml_substructure:
        key: value
        key:
            - list_item1
            - list_item2
        key: [list_item1, list_item2]
        key: {key: value, key: value}
```

Reference to names of specific structures or substructures in the YAML files are specified by ***bold_italized_colon:***.

When specifying whether something should be listed or structured or use key-value pairs, these will be underlined for emphasis.

For each specific parameter, function, variable, etc., the explanation when necessary will be in the form of
#name_of_thing# (type of thing)
This thing does/controls this and that. Accepts values like so and so. Outputs this.
For more details do the following

Reference to other parts of the documentation is specified by (see 4.1.2).

Links are specified by www.humanbrainproject.org.

2. General functionality

Currently not written.

3. YAML files

This platform employs yml (or yaml) files to specify everything necessary to build networks, simulate them, and provide outputs (a yml file is a file with the extension ".yml"). This framework offers four advantages over that of a pure python interface (such as PyNEST): 1) YML configuration files are human readable and writable, 2) they require minimal expertise to set up as long as you follow some simple syntax and limitations specified by the platform, 3) they are

flexible so you can easily change models, parts of a model, etc., and 4) they provide a way to easily handle multiple different experiments, variations, and so on, without changing the code base.

3.1 Folder and file structure

The folder structure must be the following: in your main folder (either where you put the platform files if just downloading the files, or where you want to keep all your experiment folders if you import this platform via pip install) you have an experiment folder, such as the “example” folder provided with the platform. Within that folder you have four folders; network, simulation, output, and temp.

3.1.1 network folder

Within the network folder you can have as many or as few YML files as you want, however, it’s recommended to split them into the following ones: synapsesX.yml, neuronsX.yml, layersX.yml, and connectionsX.yml. X here indicates a specific naming scheme if you want more than one such file. Each file has a specific structure with a combination of keys, key-value pairs, lists, and so on. See section 3.2 YAML syntax for more information on how to create such files specifically (else just follow the template of the example files). Parameters that deals with network properties are contained in the ***network:*** structure. The ***network:*** structure can contain the ***trash:*** substructure where you can put key-value pairs that can later be inherited by other substructures to save space and provide a one-place change-all option. The ***network:*** structure must contain the following substructures (regardless of the name of the file they’re located in): ***neurons:***, ***synapses:***, ***layers:***, and ***connections:***.

3.1.1.1 synapses.yml

The synapses.yml file is the file where all the different synapse specifications are stored. This is for creating different synapse objects (if they vary from standard template ones), that should be added your synapse models list (in NEST) for use in network construction. When creating new synapse models, the models must be listed in the ***synapses:*** substructure which is part of the ***network:*** structure. For each synapse model you want to include in your network, make a new list entry under ***synapses:*** as seen in the example file. As you can see, we have included two types of synapse models names “stat_syn” and “tso_syn”.

Specifically the synapses.yml file should look like this:

```
---
network:
     trash:
     synapses:
          -
            name: name_of_synapse
            type: type_of_base_synapse
            parameters:
                key: value
                key: value
          -
            name: another_name_of_synapse
            type: type_of_base_synapse
            parameters:
                key: value
                key: value
…
```

See the provided example files in the “example” folder for more details.

#name# (YAML parameter)
A key-value pair that contains the new name of the synapse ***type:*** you specify. If you don’t change anything from the default parameters, you don’t need to add a new synapse model to your model list. Takes: a string (name: example_name)

#type# (YAML parameter)
A key-value pair that contains the default synapse model to be used as a template for creating the new synapse model. Takes: a string (name: example_name). Note: this default model (base type) must exist in NEST. To check which ones exist, type the following in a terminal:

```
python
import nest
nest.Models()
```

#parameters# (YAML parameter)
A structure that contains structured key-value pairs. These pairs contain all the updated values for the default parameters in the default synapse model (base type) that was specified in ***type:***. Takes: structured key-value pairs. Note: the name (key) of these parameters must exist for the chosen default synapse model. To check which ones exists and what values they hold, type the following in a terminal (where “synapse” is the name of your default model you use as a base type):

```
python
import nest
nest.GetDefaults(synapse) #e.g. nest.GetDefaults(“static_synapse”)
```

3.1.1.2 neurons.yml

The neurons.yml file is the file that contains all the different neuron specifications. This if for creating different neuron objects (if they vary from the standard template ones). Neuron models are listed in the ***neurons:*** substructure which is part of the ***network:*** structure. For each neuron model you want to include in your network, make a new list entry under ***neurons:*** as seen in the example file. As you can see, we have included two types of neuron models names “model1” and “model2” as well as a sinusoidal poisson generator for delivering spikes to the network, named “Retina”.

Specifically the neurons.yml file should look like this:

```
---
network:
   trash:
   neurons:
      -
        name: name_of_neuron
        type: type_of_base_model
        parameters:
          key: value
          key: value
      -
        name: another_name_of_neuron
        type: type_of_base_model
        parameters:
          key: value
          key: value
…
```

See the provided example files in the “example” folder for more details.

#name# (YAML parameter)
A key-value pair that contains the new name of the neuron ***type:*** you specify. If you don’t change anything from the default parameters, you don’t need to add a new neuron model to your model list. Takes: a string (name: example_name)

#type# (YAML parameter)
A key-value pair that contains the default neuron model to be used as a template for creating the new neuron model. Takes: a string (name: example_name). Note: this default model (base type) must exist in NEST. To check which ones exist, type the following in a terminal:

```
python
import nest
nest.Models()
```

Note that generators such as the “sinusoidal_poisson_generator” or “step_current_generator” can be used as a neuron model.

#parameters# (YAML parameter)
A structure that contains structured key-value pairs. These pairs contain all the updated values for the default parameters in the default synapse model (base type) that was specified in ***type:***. Takes: structured key-value pairs. Note: the name (key) of these parameters must exist for the chosen default neuron model. To check which ones exists and what values they hold, type the following in a terminal (where “neuron” is the name of your default model you use as a base type):

```
python
import nest
nest.GetDefaults(neuron) #e.g. nest.GetDefaults("iaf_psc_alpha")
```

3.1.1.3 layers.yml

The layers.yml file is the file that contains all the different layer specifications. This is for defining the size and shape of your areas, as well as how many neurons you want in them, of which type, and how dense. Layer information is listed in the ***layers:*** substructure, which is part of the ***network:*** structure. One of the yml files needs to contain information about layers. Each layer consists of a group of neurons on a grid. There can be many neurons at each grid node, of different or the same type. Conceptually they will stay “on top” of each other on the grid, but otherwise not interacting. For example, if one layer contains three neuron types at each point, the layer can be considered as a stack of three layers. How you wish to connect your model dictates how you want to set it up. You can just as easily split them into separate layers.

Specifically, the layers.yml file should look like this:

```
---
network:
    trash:
    layers:
          -
             name: name_of_layer
             parameters:
                  rows: number_of_rows
                  columns: number_of_columns
                  extent: [rows_of_neurons,columns_of_neurons]
                  edge_wrap: boolean
                  elements: [name_of_neuron,name_of_neuron]
          -
             name: another_name_of_layer
             parameters:
                  rows: number_of_rows
                  columns: number_of_columns
                  extent: [rows_of_neurons,columns_of_neurons]
                  edge_wrap: boolean
                  elements: [name_of_neuron,name_of_neuron]
...
```

See the provided example files in the “example” folder for more details.

#name#
The name is simply the name of the layer, which will be used later. Takes: a string (name: example_name)

#elements#
Elements contains a list of which neuron models you want to include in the layer. Takes: listed names of neuron models

Elements:
- example_model1
- example_model2

or

Elements: [example_model1, example_model2]

This layer then contains two elements, stacked at each point of the grid (see below). You can have as many or as few (minimum one) types of elements in each layer.

#rows / columns#
The ***rows:*** and ***columns:*** key value pairs contain the size of the layer, e.g. 40 units wide and 40 units high. One unit is simply an abstract metric measuring the distance between each row and column line of the layer. Takes: integer values (rows: 10, columns: 10)

#extent#

Extent contains two list entries, referring to rows and columns, respectively. Extent says something about the number of neurons to fit on the grid described by the ***rows:*** and ***columns:***. If the same numbers are used there will be one unit distance between each neuron. See http://www.nest-simulator.org/wp-content/uploads/2014/12/Topology_UserManual.pdf for further details. Takes: double/float values
Extent:

- 10.0
- 10.0

#edge_wrap#
Edge wrap specifies whether or not the layer should wrap around, i.e. whether the top row of neurons is adjacent to the bottom row of neurons. Similarly for the left most/right most columns of neurons. Takes: boolean (edge_wrap: True). Recommended to turn this on.

3.1.1.4 connections.yml

In the connections.yml file, we specify each and every connection pattern we want, but luckily not every connection itself! The general idea is that you define a mask that every neuron in a source layer will project onto, on a target layer. Then you can specify the probability of connecting to a given neuron within that mask, delays, weights, and so on. The ***connections:*** substructure is part of the ***network:*** structure. Since there can be a lot of connections, it often makes sense to split them over multiple connections.yml files. The only thing to remember is to change the name of the ***connection_group_name:*** substructure to something unique for each file. For more details regarding connections and all the options, see http://www.nest-simulator.org/wp-content/uploads/2014/12/Topology_UserManual.pdf. Two useful key-value pairs to include (not mentioned in examples or mentioned in this document) in the ***params:*** substructure, are the ***allow_autapses:*** and ***allow_multapses:***. These control whether nodes can self connect, or connect multiple times to the same node, respectively. Both these are set to True by default.

The connections.yml file should look like this:

```
---
network:
    trash:
    connections:
         connection_group_name:
              -
                source_area: prefix_of_source_layers
                target_area: prefix_of_target_layers
                source_layer: name_of_source_layers
                target_layer: name_of_target_layers
                params:
                      sources: {model: name_of_neuron}
                      targets: {model: name_of_neuron}
                      connection_type: connectivity_style
                      synapse_model: synapse_model_to_use
                      mask:
                            mask_shape:
                                key: value
                      kernel:
                           kernel_type:
                                 key: value
                        weights: strength_of_connection_weight
                        delays:
                            delay_distribution_type:
                                 key: value
              -
                source_area: prefix_of_source_layers
                target_area: prefix_of_target_layers
                source_layer: name_of_source_layers
                target_layer: name_of_target_layers
                params:
                      sources: {model: name_of_neuron}
                      targets: {model: name_of_neuron}
                      connection_type: connectivity_style
                      synapse_model: synapse_model_to_use
                      mask:
                            mask_shape:
                                key: value
                      kernel:
                           kernel_type:
                                 key: value
                        weights: strength_of_connection_weight
                        delays:
                            delay_distribution_type:
                                 key: value
...
```

See the provided example files in the “example” folder for more details.

#source_area/target_area#
This is the prefix of where the connection is coming from or going to (source, target). This can be left empty if no prefix is needed (then use the full layer name in ***source/target_layer:***) However, if one wants to use the same connectivity scheme for two pairs of layers, one can copy/paste the connection file and just change the prefix (e.g. S_1_ -> S_2_) and the ***`connection_group_name:`***. Takes: string (source_area: S_, target_area: T_).

#sources/targets#
This specifies which neuron population the connection is coming from and going to. Here you can specify the same ***elements*** that you specified in the ***layers*** structure. Takes: structured key-value pairs (sources: {model: neuron_model}). Default is all elements.

#connection_type#
This indicates whether the connection should diverge (divergent) from a single node onto target nodes using the specified ***mask:***, or if the connection should converge onto a single node from source nodes specified by ***mask:*** (convergent). Takes: string, divergent/convergent (connection_type: divergent).

#synapse_model#
This specifies which synapse model to use. As with other models such as neurons, signal generators, etc., one can add a custom synapse type to synapse structure (see 3.1.1). Takes: string (synapse_model: static_synapse).

#mask#
This specifies what kind of mask to use for the connections diverging onto target nodes or converging from source nodes. See http://www.nest-simulator.org/wp-content/uploads/2014/12/Topology_UserManual.pdf.. The structure of the parameters and key value pairs should follow the structure in the NEST documentation. This option is optional, and has a default mask (whole layer). Takes: see link and follow structure provided in examples/NEST documentation. Default is whole layer.

#kernel#
This specifies the connection probabilities to be applied within the mask. See http://www.nest-simulator.org/wp-content/uploads/2014/12/Topology_UserManual.pdf.. The structure of the parameters and key value pairs should follow the structure in the NEST documentation. This option is optional, and has a default kernel (constant probability of 1.0). Takes: see link and follow structure provided in examples/NEST documentation. Default is connect to all nodes.

#weights#
This specifies the connection weights to be applied to each connection. Weight controls the amplification of the signal sent from one neuron to another. See http://www.nest-simulator.org/wp-content/uploads/2014/12/Topology_UserManual.pdf. The

structure of the parameters and key value pairs should follow the structure in the NEST documentation. This option is optional, and has a default weight (weight: 1.0). Takes: see link and follow structure provided in examples/NEST documentation. Default is uniform weight distribution of 1.0.

#delays#
This specifies the connection delays to be applied to each connection. Delay governs the time it takes for a signal to transmit from one neuron to the next. See http://www.nest-simulator.org/wp-content/uploads/2014/12/Topology_UserManual.pdf.. The structure of the parameters and key value pairs should follow the structure in the NEST documentation. This option is optional, and has a default value (delay: 1.0). Note: for certain distributions like the normal distribution, it's useful to also use the ***min:*** key-value with the value being set to the resolution of the simulation. Takes: see link and follow structure provided in examples/NEST documentation. Default is uniform delay distribution of 1.0 (ms).

3.1.2 Simulation folder

Within the simulation folder you can have as many or as few yml files as you want, however, it's recommended to split them into the following ones: outputs.yml, states.yml, recorders.yml, and nest_parameters.yml. Each file has a specific structure with a combination of keys, key-value pairs, lists, and so on. See section 3.2 YAML syntax for more information on how to create such files specifically (else just follow the template of the example files). Parameters that deals with simulation properties are contained in the ***simulation:*** structure. The ***simulation:*** structure can contain the ***trash:*** substructure where you can put key-value pairs that can later be inherited by other substructures to save space and provide a one-place change-all option. The ***simulation:*** structure must contain the following substructures (regardless of the name of the file they're located in): ***nest_initialization:***, ***output:***, ***states:***, ***simulation_params:***, ***recorders:***, and ***record_from:***.

3.1.2.1 nest_parameters.yml

In ***nest_initialization:*** and ***nest_other:*** substructures we specify the initial parameters that controls the engine of the NEST simulation itself, along with specific outputs, paths, and so forth. In ***nest_initalization:*** substructure, you can add anything that is accepted by:

```
nest.SetKernelStatus()
```

See nest manual for more information. In ***nest_other:*** substructure, you can currently only specify randomization options. The nest_parameters.yml file should look like the following:

```
---
simulation:
  nest_initialization:
    local_num_threads: number_of_threads
    resolution: delta_t_resolution
    dict_miss_is_error: boolean
    overwrite_files: boolean
    data_path: name_of_temp_folder
    data_prefix: prefix_temp_data
    print_time: boolean
  nest_other:
    randomize: boolean
    rand_seed: seed_number
...
```

See the provided example files in the “example” folder for more details.

#local_num_threads#
One of the parameters that is most relevant to change, this indicates how many threads to use during simulation, and should be set to as many as you want as long as it’s not more than the maximum number of threads your system can handle (usually 2 times the number of cores your processor has). Takes: integer value (local_num_threads: 16)

#resolution#
Resolution, an important parameter, specifies the delta t, or timestep, between each step of the simulation, and governs the accuracy of the solutions of differential equations. The lower the resolution, the higher the accuracy. However, note that at very small resolutions (model dependent) machine precision might be a factor, i.e. relative change in parameters too small to register leading to no change over time. Note: the resolution must be lower or equal to the minimum recording interval of your recorders, as well as the minimum delay of any connection (recording interval and minimum delay must be a multiple of your resolution). Takes: double/float value, in ms (resolution: 0.1)

#dict_miss_is_error#
This specifies if dictionary related errors are reported or not (and if the system should continue if encountering them or not). It is not recommended to have this turned off. Takes: boolean (dict_miss_is_error: True)

#overwrite_files#
This specifies whether temp files should be overwritten or not. For conserving memory/storage space, it is recommended to have this turned on. Takes: boolean(overwrite_files: True)

#data_path#

This specifies the folder name the temporary data folder placed inside your experiment folder. Takes: string (data_path: temp) <- will put temporary data in ~/experiment_folder/temp/

#data_prefix#
Prefix to your temporary files. Takes: string (data_prefix: prefix)

#print_time#
Print timestamps to terminal/console when simulating. Takes: boolean (print_time: True)

#randomize#
This specifies whether the internal NEST kernel should be randomized or set to a specific seed, i.e. if it should start with the same initial starting conditions, always. Takes: boolean (randomize: True).

#rand_seed#
This specifies the seed to be used if ***randomize:*** is True. Takes: integer value (rand_seed: 12345).

3.1.2.2 outputs.yml

The outputs.yml file specifies different options regarding the outputs from your network, such as file format and organization of the raw data, and what kind of premade plots you want.
The outputs.yml file should look like this:

```
---
simulation:
   output:
      data:
         reset: boolean
         save:
            name: name_of_run
            style: [list_of_styles]
            formats: [list_of_formats]
      graphical: [list_of_premade_plots]
...
```

See the provided example files in the “example” folder for more details.

#reset#
This specifies whether you want to delete the temporary data files generated during simulation after everything is done. They are usually not needed afterwards. Takes: boolean (reset: True)

#name#
This specifies the name of the particular run you want to save, for example you want to use the same network to simulate two different state changes, then this can be useful to set to something informative. Takes: string (name: electric_stimulation)

#style#
This specifies which style you want your raw data output to have, provided in list format. There is currently only one option, “2D”. “2D” is a neuron by time 2D matrix, where if you record from two populations of nodes in a layer, they will be appended at the bottom. For example, if you record from 900 nodes of node_type1 and 900 nodes of node_type2, simulated for 500ms with a ***resolution:*** of 0.1ms, this results in a 1800x5000 matrix. Takes: a list of styles (style: [2D])

#formats#
This specifies which format you want to save your data in. Currently only supports Matlab based files, i.e. “.mat” files. Takes: a list of formats (formats: [“.mat”])

#graphical#
This specifies which premade plots you want to have produced. Currently supports “raster” and “mean_currents”. “Raster” is a rasterplot of your data in a neuron by time format, while “mean_currents” is a line plot with the average activity (of your chosen current) over time. Both plots comes with breaker lines where your ***sim_sequence:*** switches between two states. Takes: a list of graphical output formats (graphical: [“raster”,”mean_currents”])

3.1.2.3 states.yml

States is the structure that governs the flow of the simulation, including any intermediary changes you want to do like deliver an impulse, change conductances, weights, or other things. The state structure consists of several parts, with the general structure as follows:

```
---
simulation:
    simulation_params:
        sim_sequence: [state1, state2]
    states:
        state1:
          state_length: simulation_length (double/float)
          steps: number_of_steps (integer)
          discard: boolean
          neurons:
             -
               layers: [layer1,layer2]
               populations: [model1,model2]
               modulators:
                  probability: 1.0
                if: 1.0
                properties:
                    key: value
              change:
```

```
                  key: value

            synapses:
               -
                  synapse_type: name_of_synapse_type
                  sources:
                     layers: list_of_layer_names
                     populations: list_of_elements
                  targets:
                     layers: list_of_layer_names
                     populations: list_of_elements
                  change:
                     key: value
...
```

See the provided example files in the “example” folder for more details.

#sim_sequence#
The sim_sequence wants a list of named states. One state can be repeated if you want to run it several times. The names of states will be used later (and in a future version also be implemented in automatic plotting) when you specify each state in ***states:***. Takes: list of state names (sim_sequnce: [state1, state2, …])

#state1#
This is a substructure within ***states:*** whose name must match one of those in ***sim_sequence:***.

#state_length#
This is the length of the simulation of this specific state, in ms. Takes: double/float (state_length: 100.0)

#steps#
How many steps you want to simulate over, and also apply any changes over. Steps can be a minimum of 1. For example, if you want to increase a neuron property gradually, you can set steps to any number you’d like, and then it will apply a linear change of the parameter in the way and direction you want. Takes: integer (steps: 2)

#discard# ***currently unsupported***
This specifies whether you want to record during this period or not. This is useful to save memory/disk-space when simulating, and should be used in states where the network is settling into a stable pattern, or during a transition you’re not interested in. Note: data will be appended together at the cut if discard is set to True for a non-first state. Takes: boolean (discard: False).

#neurons/synapses#

The ***neurons:*** and ***synapses:*** substructures contain listed structures for changes applying to neurons and synapses, respectively.

#layers#
List of layers you want to apply a change to. Note: if you want to make sweeping changes, you can use parts of a layer name, and the platform will find all layers matching that. For example, you have fifty layers names layer1, layer2, … . Then you can set layers: [layer] and all layers will be selected. Clever naming of layers in the ***layers:*** substructure in the ***network:*** structure (layers.yml), can make this part quicker. Takes: list of layer names or parts of layer names (layers: [layer1, layer2]

#populations#
List of populations, within layers, you want to apply a change to. Note: you can use part of element names, just as in ***layers:***. Takes: list of element names, or parts of element names (populations: [pop1, pop2])

#modulators#
Modulators are specific rules to apply when selecting synapses or neurons to apply a change to. Initially, the platform finds all nodes/synapses specified in layers/populations/sources/targets, and then modulates that selection further. There are two main ways of modulation. One is ***probability:*** which controls random selection of elements, and the other is ***if:*** which uses logicals to decide inclusion or exclusion. Takes: key-value pair (probability: 0.5), and/or, ***if:*** value, and ***properties:***.

#probability#
This decides the probability of inclusion, i.e. 50% = 0.5. Takes: double/float (probability: 0.5)

#if#
This decides which rule to apply for the properties. Takes: integer (if: 1), where 0=not equals, 1= equals, 2=bigger than, 3=smaller than. If the ***if:*** key-value pair exists, ***properties:*** must also exist. Logicals are specified with for example if neuron property X is bigger than the specified property in ***properties:***.

#properties#
This substructure contains key-value pairs, with properties that will be used as a basis for selection. For a list of possible key-value pairs, write the following in terminal:

```
python
import nest
nest.GetDefaults(neuron/synapse)
```

#synapse_type#

This specifies which synapse types are to be changed. This can be left empty if no synapses are to be changed. Takes: string of synapse model (synapse_type: static_synapse).

#sources#
This substructure contains key-value pairs to specify the source of the connections you want to change.

#targets#
This substructure contains key-value pairs to specify the targets of the connections you want to change.

#layers#
See ***layers:*** above.

#populations#
See ***populations:*** above.

#change#
Here you specify what to change. ***Change:*** is a substructure within each list item underneath ***neurons:*** or ***synapses:***. It takes key-value pairs, specifying which properties to change. However, one can apply a change in multiple ways, and so any key-value pair has a list as its value. The first item in the list is the numeric change, i.e. 12.0, and the second item is the type of change, being either “a” for additive change, “c” for constant change, “r” for random change, “p” for percent change, and finally “code” for custom change. These changes, can be scaled over ***steps:*** so that a change can be implemented gradually (for “code” this needs to be specified manually). See the *change* section below for details. Note: when changing synapses you are not changing the synapse model but specific connections between nodes, which means you can for example increase the weight of only synapses with certain properties such as their sources and targets. Takes: key-value pairs (V_m: [20.0,a])
For a list of possible key-value pairs, write the following in terminal:

```
python
import nest
nest.GetDefaults(neuron/synapse)
```

change
Changes can be either additive, constant, random, percent, or code based. For example:
C = constant, i.e. V_m: [-70.0, c] will set the membrane potential to -70.0 over X steps.
A = additive, i.e. V_m: [-70.0, a] will add -70.0 to the membrane potential over X steps.
R = random between two values (needs list format), i.e. V_m: [[-70.0,-50.0], r] will set membrane potential to a random value between 0 and -70.0 (will do this one timer per step).
P = percent, i.e. V_m: [1.5, p] will increase the membrane potential by a factor of 1.5, or 50%.

Code = custom code, i.e. V_m: ["[z*np.random.choice([-1,1],1)) for z in range(len(value1))]" ,code]
The code type change is for those times when you want to do something special with a change. It will accept whatever you put into it, as long as it follows some limitations. A custom code has available only a list termed value1, number of steps to apply a change over, and number of steps taken. Value1 is a list with all the parameter values for all the neurons/synapses you wanted to apply the change to, for example a list of all the membrane potentials of all the neurons specified by the layers and populations parameters. In addition, you have access to the numpy package, imported as “np”. The above code would result in setting the membrane potential of all selected neurons to z multiplied with 1 or -1, where z is determined by the neuron number, i.e. the thousandth neuron would have its membrane potential set to either -1000 or 1000. The sequence of neurons are as follows: given a rectangular layer, neuron 1 is the top left, neuron 2 is top second left, and so on. At the end of the top row, the next neuron is then second top row left, and so on. When at the end of a layer, the next neuron would be top left of the next layer or population, where layer sequence is specified in the ***layers:*** structure of the yml files, while populations are specified within the ***layers: elements:*** structure of ***layers:*** structure of the yml files. Remember that the result of the code needs to output a list with new values that is equally long as the original list (value1). There are no checks to see if your code is valid python code.

3.1.2.4 recorders.yml

In the recorders.yml file we specify which recorders should be created and which layers and neurons they should record from. This is the basis for all the output of the model, and so should be selected carefully as larger models quickly generates obscene amounts of data. Recorder infomation is the only file in the “simulation” folder that has its parameters within the ***network:*** structure. There exists multiple types of recorders, however the most general and flexible of them is the “multimeter”, which can be configured to record most of the same things as the other recorders. Recording of synapse weights is currently not supported. Within ***network:*** structure is the ***recorders:*** substructure, which contains the ***types:*** and ***record_from:*** substructures.
Types: contains different recorder specifications, while ***record_from:*** contain information where to connect these recorders.
The general structure is as follows:

```
---
network:
 recorders:
  types:
    -
        type: recorder_type
        name: name_of_recorder
        params:
             interval: recording_interval
             record_from: record_from_properties
             record_to: storage_type
    -
        type: recorder_type
        name: name_of_recorder
        params:
             interval: recording_interval
             record_from: record_from_properties
             record_to: storage_type

  record_from:
    -
        layer: name_of_target_layer
        population: name_of_target_node_population
        recorder: name_of_recorder
    -
        layer: name_of_target_layer
        population: name_of_target_node_population
        recorder: name_of_recorder
...
```

See the provided example files in the “example” folder for more details.

#type#
This governs which type of recorder it should be. A multimeter is the most standard one as it can handle most kinds of recordables. Takes: string (type: multimeter)

#name#
The name of this particular recorder, useful for later. Takes: string (name: my_multimeter)
#interval#
This is the recording interval in ms. The lower this value is, the higher precision, but increased memory/space is needed. Note: this value cannot be lower than ***resolution:*** as defined in ***nest_initialization:***. Takes: double/float (interval: 0.1)

#record_from#

This specifies which properties to record from, from the neuron in question. Here you can make a list if you want to record from multiple properties. For a list of possible recordably properties, type in terminal:

```
Python
import nest
nest.GetDefaults(model_name)["recordables"]
#model_name="iaf_neuron"
```

record_from: can take multiple arguments and so uses the list format. Takes: list of recordables (record_from: ["V_m","I_NaP"])

#record_to#
This specifies if you want to record to memory and/or file. Takes: list of record to options, "memory" and/or "file" (record_to: [memory])

#layer#
This the layer you want to record from. Use a name from the ***layers:*** substructure. Currently only supports recording from one layer. Takes: string (layer: layer_name)

#population#
This is the name of neuron model you want to record from, from within a layer. This population of nodes must be in the target layer specified above. Currently only supports recording from one population type within a layer. Takes: string (population: population_name)

#recorder#
This specifies which recorder to use from the ones listed in ***types:***. Make sure that the recorder can record from the population in question, i.e. that the recorders ***record_from:*** property is in the list of that neuron model's recordables. Takes: string (recorder: recorder_name).

3.1.3 Output folder

In the output folder, the output of the model will appear. Within the output folder, there will be a folder called "name" which is specified by the ***name:*** property in "outputs.yml". This specifies a specific instance of a simulation, for example if you have a model that you want to run one simulation using electric stimulation of a layer, while another where you want to change a time constant parameter of neurons within a layer, you could specify a different name for those two (electric_stim, tau_change), and only change the ***sim_sequence:*** parameter in "states.yml".

3.1.4 Temp folder

The temp folder will contain the temporary files (if recorders record to file) of the activity in your simulation. At the end of the simulation (if recorders record to file), these files will be loaded one recorder at a time, and output will be generated and put in the output folder. These files are

usually not needed after output has been generated, and can be removed after a simulation by setting ***reset:*** in “nest_parameters.yml” to True.

3.2 YAML syntax

The YAML syntax is fairly basic, however, see http://docs.ansible.com/ansible/latest/YAMLSyntax.html for a more thorough walkthrough. For the purposes of this documentation, only the necessary is explained.

A YAML file ends with “.yml”. This file can be opened in any text editor. It’s main element is a structure, with substructures, lists, key-value pairs, and so on.

```
--- ### YAML files starts with this (for this platform)
### this is a comment
structure:
   substructure:
      key1: value
      key2: [list]
      structure:
        substructure:
         - list item
         - list item
… ### YAML files end with this (for this platform)
```

In the above code block, there’s an example yaml file. Structure, or dictionary if you are familiar with python/struct if matlab, is a container (called dictionary in YAML) that contains lists, other dictionaries (here called substructures), and so on. In the example above, the yaml file contains a structure called ***structure:*** which has a substructure called ***substructure:***. Each structure is actually just a key-value pair but with nested levels instead of a simple value. For the purposes of this doc. it will be referred to as structure for top level keys, and substructure for those values that are keys beneath those structures.

Nestedness is controlled by indentation. So, in the example above, we have a key called structure, that contains a value named substructure, which contains three keys called key1, key2, structure, and the key structure again contains a key called substructure, which contains a list with two items. There’s only two important things to consider, 1) use spaces for indentation, and 2) be consistent with how many spaces you use to indent different levels.

The concept can be summarized as containers containing values, lists of values, lists of containers, or more containers. As such, YAML is deeply nested. Clever use of these files can however bring large time savings when developing a network model to simulate, changing a model, or expand a model.

Finally, YAML files can use inheritance. This is when one container inherits the contents of another container. This is controlled by the & anchor symbol, and the * reference symbol, as such:

```
---
shopping_list:
  base: &anchor1
    bread: 10
    milk: 2
    cheese: 3
  monday:
    <<: *anchor1
    eggs: 40
    spinach: 2
  tuesday:
    <<: *anchor1
    meat:
      ham: 2
      bacon: 1
```

For employing an anchor (standard) from which to inherit properties, use “trash” at 1 sublevel below the main structure, for example “network:”, as seen in “example/network/neurons.yml”.

To iterate: a key-value pair is a key: value, where the value can be a number (int/float), a string (use “ “ if special characters like space), a list ([list1, list2, list3]), or more key-value pairs (key: {key: value, key: value}.
For both lists, and key-value pairs, the one can write them as [list_item1, list_item2] and {key1:value1, key2:value2}, or nest below, as you can see in the above examples. A list item can then contain more key-value pairs, lists, and so on, and the value part of a key-value pair can contain other key-value pairs, whose values are lists or more key-value pairs, and so on.

4. Appendix

4.1 Changelog

V0.1

- Initial release

4.2 Copyright

4.3 Acknowledgments

Thanks to Thierry Nieus (2), Ricardo Murphy (1), Hans E. Plesser (3,4), Sean Hill (5,6), Will G. Mayner (7), Tom Bugnon (7), Bjørn Juel (1), and Johan Storm (1).

(1) Brain Signalling Group, Department of Physiology, Institute of Basic Medical Sciences, University of Oslo, Oslo, Norway
(2) Department of Biomedical and Clinical Sciences "Luigi Sacco" University of Milan, Milan, Italy
(3) Faculty of Science and Technology, Norwegian University of Life Sciences, Ås, Norway
(4) Institute for Neuroscience (INM-6), Jülich Research Centre, Jülich, Germany
(5) Krembil Centre for Neuroinformatics, CAMH, Toronto, Canada
(6) Blue Brain Project, EPFL, Geneva, Switzerland
(7) Wisconsin Institute for Sleep and Consciousness, University of Wisconsin-Madison, USA

4.4 Dependencies

Implemented for NEST version 2.12 and Python 3.x.

Dependencies: Numpy, Matplotlib, Scipy, Pandas, os, glob, sys, yaml, collections, nest, nest.topology, copy, joblib, multiprocessing.

4.5 Citation information

Citation: please contact sevenius.nilsen@gmail.com if you want to use this platform or the accompanying “ht_model” in commercial or scientific work. Educational purposes are exempt.