

GeNN

1.1

Generated by Doxygen 1.8.7

Thu Dec 4 2014 14:24:23

Contents

Chapter 1

Main Page

GeNN is a software package to enable neuronal network simulations on NVIDIA GPUs by code generation. Models are defined in a simple C-style API and the code for running them on either GPU or CPU hardware is generated by GeNN. GeNN can also be used through external interfaces. Currently there are prototype interfaces for Spine↔Creator and SpineML and for [Brian2](#).

GeNN is currently developed and maintained by

[James Turner](#) ([contact James](#))

[Dr. Esin Yavuz](#) ([contact Esin](#))

[Prof. Thomas Nowotny](#) ([contact Thomas](#))

1.1 Contents

[1 Installation](#)

[2 Quickstart](#)

[3 Examples](#)

[4 List of GeNN Variables](#)

[5 User Manual](#)

The development of GeNN is partially supported by the [EPSRC](#) (grant number [EP/J019690/1 - Green Brain Project](#)).

Note

This documentation is under construction. If you cannot find what you are looking for, please contact the project developers.

[Next](#)

Chapter 2

Installation

You can download GeNN either as a zip file of a stable release or a snapshot of the most recent stable version or the unstable development version using the Git version control system.

2.1 Downloading a release

Point your browser to <https://github.com/genn-team/genn/releases> and download a release from the list by clicking the relevant source code button. Note that GeNN is only distributed in the form of source code due to its code generation design. Binary distributions would not make sense in this framework and are not provided. After downloading continue to install GeNN as described in the install section below.

2.2 Obtaining a Git snapshot

If it is not yet installed on your system, download and install Git (<http://git-scm.com/>). Then clone the GeNN repository from Github

```
git clone https://github.com/genn-team/genn.git
```

The github url of GeNN in the command above can be copied from the HTTPS clone URL displayed on the GeNN Github page (<https://github.com/genn-team/genn>).

This will clone the entire repository, including all open branches. By default git will check out the master branch which contains the source version upon which the latest release is based. If you want the most recent (but unstable) development version (which may or may not be fully functional at any given time), checkout the development branch

```
git checkout development
```

There are other branches in the repository that are used for specific development purposes and are opened and closed without warning.

Alternatively to using git you can also download the full content of GeNN sources clicking on the "Download ZIP" button on the bottom right of the GeNN Github page (<https://github.com/genn-team/genn>).

2.3 Installing GeNN

Installing GeNN comprises a few simple steps to create the GeNN development environment.

(i) If you have downloaded a zip file, unpack GeNN.zip in a convenient location. Otherwise enter the directory where you downloaded the Git repository.

(ii) Define the environment variable "GENN_PATH" to point to the main GeNN directory, e.g. if you extracted/downloaded GeNN to /usr/local/GeNN, then you can add "export GENN_PATH=/usr/local/GeNN" to your login script (e.g. .profile or .bashrc. If you are using WINDOWS, the path should be a windows path as it will be interpreted by the Visual C++ compiler cl, and environment variables are best set using setx in a Windows cmd window. To do so, open a Windows cmd window by typing cmd in the search field of the start menu, followed by the enter key. In the cmd window type

```
setx GENN_PATH "C:\Users\me\GeNN"
```

where C:\Users\me\GeNN is the path to your GeNN directory.

(iii) Add \$GENN_PATH/lib/bin to your PATH variable, e.g.

```
export PATH=$PATH:$GENN_PATH/lib/bin
```

in your login script, or in windows,

```
setx PATH=%PATH%;%GENN_PATH%\lib\bin
```

(iv) If you haven't installed CUDA on your machine, obtain a fresh installation of the NVIDIA CUDA toolkit from <https://developer.nvidia.com/cuda-downloads>

(v) Set the CUDA_PATH variable if it is not already set by the system, by putting

```
export CUDA_PATH=/usr/local/cuda
```

in your login script (or, if CUDA is installed in a non-standard location, the appropriate path to the main CUDA directory). For most people, this will be done by the CUDA install script and the default value of /usr/local/cuda is fine. In Windows, use setx to set this variable,

```
setx CUDA_PATH
```

This normally completes the installation.

Depending on the needs of your own projects, e.g., dependencies on other libraries or non-standard installation paths of libraries used by GeNN, you may want to modify Makefile examples under \$GENN_PATH/userproject/xxx_↵_project and \$GENN_PATH/userproject/xxx_project/model to add extra linker-, include- and compiler-flags on a per-project basis, or modify global default flags in \$GENN_PATH/lib/include/makefile_↵_common.mk.

For all makefiles there are separate makefiles for Unix-style operating systems (GNUmakefile) such as Linux or MacOS and for Windows (WINmakefile).

If you are using GeNN in Windows, you can use make.bat to build examples which will attempt to setup your development environment by executing vcvarsall.bat which is part of every Visual Studio distribution. For this to work properly, GeNN must be able to locate the Visual Studio install directory, which should be contained in the VS_PATH environment variable. You can set this variable by hand if it is not already set by the Visual C++ installer by typing

```
setx VS_PATH "C:\Program Files (x86)\Microsoft Visual Studio 10.0"
```

Note

- The exact path and name of Visual C++ installations will vary between systems.
- Double quotation marks like in the above example are necessary whenever a path contains spaces.

GeNN also has experimental CYGWIN support. However, with the introduction of native Windows support in GeNN 1.1.3, this is not being developed further and should be considered as deprecated.

2.4 Testing Your Installation

To test your installation, follow the example in the [Quickstart section](#).

[Previous](#) | [Top](#) | [Next](#)

Chapter 3

Quickstart

GeNN is based on the idea of code generation for the involved GPU or CPU simulation code for neuronal network models but leaves a lot of freedom how to use the generated code in the final application. To facilitate the use of GeNN on the background of this philosophy it comes with a number of complete examples containing both the model description code that is used by GeNN for code generation and the "user side code" to run the generated model and save the results. Running these complete examples should be achievable in a few minutes. The necessary steps are described below.

3.1 Running an Example Model in Unix

In order to get a quick start and run a provided model, open a shell, navigate to `GeNN/tools` and type

```
make
```

This will compile additional tools for creating and running example projects. For a first complete test, the system is best used with a full driver program such as in the [Insect Olfaction Model](#) example:

```
tools/generate_run [CPU/GPU] [#AL] [#KC] [#LH] [#DN] [gscale] [OUTNAME] [MODEL] [DEBUG] [FTYPE] [REUSE].
```

To compile `generate_run.cc`, navigate to the `userproject/MBody1_project` directory and type

```
make all
```

This will generate an executable that you can invoke with, e.g.,

```
./generate_run 1 100 1000 20 100 0.0025 test1 MBody1 0 FLOAT 0
```

which would generate and simulate a model of the locust olfactory system with 100 projection neurons, 1000 Kenyon cells, 20 lateral horn interneurons and 100 output neurons in the mushroom body lobes.

The tool `generate_run` will generate connectivity matrices for the model `MBody1` and store them into files, compile and run the model on the GPU using these files as inputs and output the resulting spiking activity. To fix the GPU used, replace the first argument `1` with the device number of the desired GPU plus 2, e.g., `2` for GPU 0. All input and output files will be prefixed with `test1` and will be created in a sub-directory with the name `test1_output`. The third to last parameter `0` will switch the debugging mode off, `1` would switch it on. More about debugging in the debugging section. The parameter `FLOAT` will run the model in float (single precision floating point), using `DOUBLE` would use double precision. The last parameter regulates whether previously generated files for connectivity and input should be reused (`1`) or files should be generated anew (`0`).

The `MBody1` example is already a highly integrated example that showcases many of the features of GeNN and how to program the user-side code for a GeNN application. More details in the [User Manual](#)

3.2 Running an Example Model in Windows

All interaction with GeNN programs are command-line based and hence are executed within a `cmd` window. Open a `cmd` window and navigate to the `userprojects\tools` directory.

```
cd %GENN_PATH%\userprojects\tools
```

Then type

```
make.bat all
```

to compile a number of tools that are used by the example projects to generate connectivity and inputs to model networks.

Then navigate to the `MBody1_project` directory.

```
cd ..\MBody1_project
```

By typing

```
make.bat all
```

you can compile the `generate_run` engine that allows to run a [Insect Olfaction Model](#) model of the insect mushroom body:

```
tools/generate_run [CPU/GPU] [#AL] [#KC] [#LH] [#DN] [gscale] [DIR] [EXE] [MODEL] [DEBUG OFF/ON].
```

To invoke `generate_run.exe` type, e.g.,

```
generate_run.exe 1 100 1000 20 100 0.0025 test1 MBody1 0
```

which would generate and simulate a model of the locust olfactory system with 100 projection neurons, 1000 Kenyon cells, 20 lateral horn interneurons and 100 output neurons in the mushroom body lobes.

The tool `generate_run.exe` will generate connectivity matrices for the model `MBody1` and store them into files, compile and run the model on an automatically chosen GPU using these files as inputs and output the resulting spiking activity. To fix the GPU used, replace the first argument `1` with the device number of the desired GPU plus 2, e.g., `2` for GPU 0. All input and output files will be prefixed with `test1` and will be created in a sub-directory with the name `test1_output`. The last parameter `0` will switch the debugging mode off, `1` would switch it on. More about debugging in the debugging section.

The `MBody1` example is already a highly integrated example that showcases many of the features of GeNN and how to program the user-side code for a GeNN application. More details in the User Manual

3.3 How to use GeNN for New Projects

Creating and running projects in GeNN involves a few steps ranging from defining the fundamentals of the model, inputs to the model, details of the model like specific connectivity matrices or initial values, running the model and analyzing or saving the data.

The most common way to use GeNN is to create or modify a program such as [userproject/MBody1↔project/generate_run.cc](#) that wraps around other programs that are used for each of the necessary steps listed above. In more detail, what `generate_run` and similar programs do is:

1. To use other tools (programs) to generate connectivity matrices and store them into files.
2. To build the source code of a model simulation using GeNN. In the example of the `MBody1_project` this entails writing neuron numbers into [userproject/include/sizes.h](#), and executing

```
buildmodel.sh MBody1 [DEBUG OFF/ON]
```

The `buildmodel.sh` script compiles the installed GeNN code generator in conjunction with the user-provided model description (see [below](#)), in this example `model/MBody1.cc`. It then executes the GeNN code generator to generate the complete model simulation code for the MBody1 model.

3. To compile the generated model code, that can be found in `model/MBody1_CODE/` by invoking `make clean && make` in the `model` directory. It is at this stage that GeNN generated model simulation code is combined with user-side run-time code, in this example `classol_sim.cu` (classify-olfaction-simulation) which uses the `map_classol` (map-neuron-based-classifier-olfaction) class.
4. To finally run the resulting stand-alone simulator executable, in the MBody1 example `classol_sim` in the `model` directory.

The `generate_run` tool is only a suggested usage scenario of GeNN. Alternatively, users can manually execute the four steps above or integrate GeNN with development environments of their own choice.

Note

The usage scenario described was made explicit for Unix environments. In Windows the setup is essentially the same except for the usual operating system dependent syntax differences, e.g. the build script is named `buildmodel.bat`, compilation of the generated model simulator would be `make.bat clean && make.bat all`, or, `nmake /f WINmakefile clean && nmake /f WINmakefile all`, and the resulting executable would be named `classol_sim.exe`.

GeNN comes with several example projects which showcase how to use its features. The MBody1 example discussed above is one of the many provided examples that are described in more detail in [Example projects](#).

3.4 Defining a New Model in GeNN

According to the workflow outlined above, there are several steps to be completed to define a neuronal network model.

1. The neuronal network of interest is defined in a model definition file, e.g. `Example1.cc`.

Note

GeNN follows a convention in which C/C++ files end with `.cc` and the model definition file will only be recognized by the `buildmodel.sh` or `buildmodel.bat` scripts if it follows this convention and ends on `.cc`.

2. Within the the model definition file `Example1.cc`, the following tasks need to be completed:
 - a) The time step `DT` needs to be defined, e.g.

```
#define DT 0.1
```

Note

All provided examples and pre-defined model elements in GeNN work with units of mV, ms, nF and μ S. However, the choice of units is entirely left to the user if custom model elements are used.

- b) The GeNN files [modelSpec.h](#) and [modelSpec.cc](#) need to be included,

```
#include "modelSpec.h"
#include "modelSpec.cc"
```

- c) The values for initial variables and parameters for neuron and synapse populations need to be defined, e.g.

```
float myPOI_p[4]= {
    0.1,          // 0 - firing rate
    2.5,          // 1 - refractory period
    20.0,         // 2 - Vspike
    -60.0         // 3 - Vrest
};
```

would define the (homogeneous) parameters for a population of Poisson neurons.

Note

The number of required parameters and their meaning is defined by the neuron or synapse type. Refer to the [User Manual](#) for details.

If heterogeneous parameter values are needed for any particular population of neurons (synapses), a new neuron (synapse) type needs to be defined in which these parameters are defined as variables rather than parameters. See the [User Manual](#) for how to define new neuron (synapse) types.

d) the actual network needs to be defined in the form of a function `modelDefinition`, i.e.

```
void modelDefinition(NNmodel &model);
```

Note

The name `modelDefinition` and its parameter of type `NNmodel&` are fixed and cannot be changed if GeNN is to recognise it as a model definition.

`MBody1.cc` shows a typical example of a model definition function. In its core it contains calls to `model.addNeuronPopulation` and `model.addSynapsePopulation` to build up the network. For a full range of options for defining a network, refer to the [User Manual](#).

3. The programmer defines her own "user-side" modeling code similar to the code in `userproject/MBody1_project/model/map_classol.*` and `userproject/MBody1_project/model/classol*_sim.*`. In this code,

a) she defines the connectivity matrices between neuron groups. (In the `MBody1` example those are read from files). Refer to the [User Manual](#) for the required format of connectivity matrices for dense or sparse connectivities.

b) she defines input patterns (e.g. for Poisson neurons like in the `MBody1` example) or individual initial values for neuron and/or synapse variables.

Note

The initial values given in the `modelDefinition` are automatically applied homogeneously to every individual neuron or synapse in each of the neuron or synapse groups.

c) she uses `stepTimeGPU(...)` to run one time step on the GPU or `stepTimeCPU(...)` to run one on the CPU. (both GPU and CPU versions are always compiled).

Note

However, mixing CPU and GPU execution does not make too much sense. Among other things, The CPU version uses the same host side memory where to results from the GPU version are copied, which would lead to collisions between what is calculated on the CPU and on the GPU (see next point). However, in certain circumstances, expert users may want to split the calculation and calculate parts (e.g. neurons) on the GPU and parts (e.g. synapses) on the CPU. In such cases the fundamental kernel and function calls contained in `stepTimeXXX` need to be used and appropriate copies of the data from the CPU to the GPU and vice versa need to be performed.

d) she uses functions like `copyStateFromDevice()` etc to transfer the results from GPU calculations to the main memory of the host computer for further processing.

e) she analyzes the results. In the most simple case this could just be writing the relevant data to output files.

Chapter 4

Examples

GeNN comes with a number of complete examples. At the moment, there are seven such example projects provided with GeNN.

4.1 Single compartment Izhikevich neuron(s)

This is a minimal example, with only one neuron population (with more or less neurons depending on the command line, but without any synapses). The neurons are Izhikevich neurons ? with homogeneous parameters across the neuron population. The model can be used by navigating to the `userproject/OneComp_project` directory and entering a command line

```
./generate_run <CPU/GPU> <n> <OUTNAME> <MODEL> <DEBUG> <FTYPE> <REUSE>.
```

All parameters are mandatory and signify:

- CPU/GPU: Choose whether to run the simulation on CPU (0) or GPU (1).
- n: Number of neurons
- OUTNAME: The base name of the output location and output files
- MODEL: The name of the model to execute, as provided this would be `OneComp`
- DEBUG: Whether to start in debug mode (1) or normally (0)
- FTYPE: Floating point type `FLOAT` or `DOUBLE`
- REUSE: whether to re-use input files

This would create `n` tonic spiking Izhikevich neuron(s) with no connectivity, receiving a constant, identical 4 nA input current.

For example, navigate to the `userproject/OneComp_project` directory and type

```
make all
./generate_run 1 1 Outdir OneComp_sim OneComp 0 FLOAT 0
```

to model a single neuron which output will be saved in the `Outdir_output` directory.

4.2 Izhikevich Network Driven by Poisson Input Spike Trains:

In this example project there is again a pool of non-connected Izhikevich model neurons ? that are connected to a pool of Poisson input neurons with a fixed probability.

The model can be compiled by navigating to the `userproject\PoissonIzh_project` directory and typing

```
make all
```

Subsequently it can be invoked using the following command line

```
./generate_run <CPU/GPU> <#POISSON> <#IZHIKEVICH> <PCONN> <GSCALE> <OUTNAME> <MODEL> <DEBUG  
OFF/ON> <FTYPE> <REUSE>
```

All parameters are mandatory and signify:

- CPU/GPU: Choose whether to run the simulation on CPU (0) or GPU (1).
- #POISSON: Number of Poisson input neurons
- #IZHIKEVICH: Number of Izhikevich neurons
- PCONN: Probability of connections
- OUTNAME: The base name of the output location and output files
- MODEL: The name of the model to execute, as provided this would be `PoissonIzh`
- DEBUG: Whether to start in debug mode (1) or normally (0)
- FTYPE: Floating point type `FLOAT` or `DOUBLE`
- REUSE: whether to re-use input files

For example, navigate to the `userproject/PoissonIzh_project` directory and type

```
./generate_run 1 100 10 0.5 2 Outdir PoissonIzh 0 DOUBLE 0
```

This will generate a network of 100 Poisson neurons connected to 10 Izhikevich neurons with a 0.5 probability. The same example network can be used with sparse connectivity (i.e. sparse matrix representations for the connectivity within GeNN) by using the keyword `SPARSE` in the `addSynapsePopulation` instead of `DENSE` in [PoissonIzh.cc](#) and by uncommenting the lines following the comment `//SPARSE CONNECTIVITY` in the file `PoissonIzh.cu`. In this example the model would be simulated with double precision variables and input files and connectivity would not be reused from earlier runs.

4.3 Pulse-coupled Izhikevich Network

This example model is inspired by simple thalamo-cortical network of Izhikevich ? with an excitatory and an inhibitory population of spiking neurons that are randomly connected.

The model can be built by navigating to the `userproject/Izh_Sparse_project` directory and typing

```
make all
```

It can then be used as

```
./generate_run <CPU/GPU> <#N> <#CONN> <GSCALE> <OUTNAME> <MODEL> <DEBUG OFF/ON>  
<FTYPE> <REUSE>
```


All parameters are mandatory and signify:

- CPU/GPU: Choose whether to run the simulation on CPU (0) or GPU (1).
- #N: Number of neurons
- #CONN: Number of connections per neuron
- 'GSCALE': General scaling of synaptic conductances
- OUTNAME: The base name of the output location and output files
- MODEL: The name of the model to execute, as provided this would be `Izh_sparse`
- DEBUG: Whether to start in debug mode (1) or normally (0)
- FTYPE: Floating point type `FLOAT` or `DOUBLE`
- REUSE: whether to re-use input files

The model creates a pulse-coupled network ? with 80% excitatory 20% inhibitory neurons, each connecting to #CONN neurons using the sparse matrix connectivity methods of GeNN.

For example, typing

```
./tools/generate_izhikevich_network_run 1 10000 1000 1 Outdir Izh_sparse 0 FLOAT 0
```

generates a random network of 8000 excitatory and 2000 inhibitory neurons which each have 1000 outgoing synapses to randomly chosen post-synaptic target neurons. The synapses are of a simple pulse-coupling type. The results of the simulation are saved in the directory `Outdir_output`, debugging is switched off, and the connectivity is generated afresh (rather than being read from existing files).

Note

If connectivity were to be read from files, the connectivity files would have to be in the `inputfiles` sub-directory and be named according to the names of the synapse populations involved, e.g., `gIzh_sparse_ee (<variable name>="">=g <model name>="">=Izh_sparse_<synapse population>=_ee)`. These name conventions are not part of the core GeNN definitions and it is the privilege (or burden) of the user to find their own in their own versions of `generate_XXX_run`.

4.4 Izhikevich network with delayed synapses

This example project demonstrates the feature of synaptic delays in GeNN. It creates a network of three Izhikevich neuron groups, connected all-to-all with short, medium and long delay synapse groups. Neurons in the output group only spike if they are simultaneously innervated by the input neurons, via synapses with long delay, and the interneurons, via synapses with shorter delay.

To run this example project, navigate to `userproject/SynDelay_project` and type, e.g.,

```
buildmodel SynDelay
make clean && make
./bin/release/syn_delay 1 output
```

4.5 Insect Olfaction Model

This project implements the insect olfaction model by Nowotny et al. [1] that demonstrates self-organized clustering of odours in a simulation of the insect antennal lobe and mushroom body. As provided the model works with conductance based Hodgkin-Huxley neurons [2] and several different synapse types, conductance based (but pulse-coupled) excitatory synapses, graded inhibitory synapses and synapses with a type of STDP rule.

To explore the model navigate to `userproject/MBody1_project/` and type

```
make all
./generate_run
```

This will show you the command line parameters that are needed,

```
tools/generate_run <CPU/GPU> <#AL> <#KC> <#LHI> <#DN> <GSCALE> <OUTNAME> <MODEL> <DEBUG> <FTYPE> <
REUSE>
```

All parameters are mandatory and signify:

- CPU/GPU: Choose whether to run the simulation on CPU (0) or GPU (1).
- #AL: Number of neurons in the antennal lobe (AL), the input neurons to this model
- #KC: Number of Kenyon cells (KC) in the "hidden layer"
- #LH: Number of lateral horn interneurons, implementing gain control
- #DN: Number of decision neurons (DN) in the output layer
- GSCALE: A general rescaling factor for synaptic strength
- OUTNAME: The base name of the output location and output files
- MODEL: The name of the model to execute, as provided this would be `MBody1`
- DEBUG: Whether to start in debug mode (1) or normally (0)
- FTYPE: Floating point type `FLOAT` or `DOUBLE`
- REUSE: whether to re-use input files

The tool `generate_run` will generate connectivity files for the model `MBody1`, compile this model for the CPU and GPU and execute it. The command line parameters are the numbers of neurons in the different neuropils of the model and an overall synaptic strength scaling factor. A typical call would be, e.g.,

```
../../../../tools/generate_run 1 100 1000 20 100 0.0025 test1 MBody1 0 FLOAT 0
```

which would generate a model, and run it on the GPU (first parameter), with 100 antennal lobe neurons, 1000 mushroom body Kenyon cells, 20 lateral horn interneurons and 100 mushroom body output neurons. All output files will be prefixed with `test1` and stored in `test1_output`. The model that is run is defined in `model/MBody1.cc`, debugging is switched off, the model would be simulated using float (single precision floating point) variables and parameters and the connectivity and input would be generated afresh for this run.

As provided, the model outputs a file `test1.out.st` that contains the spiking activity observed in the simulation, where there are two columns in this ASCII file, the first one containing the time of a spike and the second one the ID of the neuron that spiked. Users of matlab can use the scripts in the `matlab` directory to plot the results of a simulation. For more about the model itself and the scientific insights gained from it see [1].

4.6 Insect Olfaction Model with User-Defined Types

This examples recapitulates the exact same model as [Insect Olfaction Model](#) above but with user-defined model types for neurons and synapses throughout. It is run the same way as [Insect Olfaction Model](#), e.g.

```
../../../../tools/generate_run 1 100 1000 20 100 0.0025 test1 MBody_userdef 0 FLOAT 0
```

But the way user-defined types are used should be very instructive to advanced users wishing to do the same with their models.

[Previous](#) | [Top](#) | [Next](#)

Chapter 5

Release Notes for GeNN v2.0

Version 2.0 of GeNN comes with a lot of improvements and added features, some of which have necessitated some changes to the structure of parameter arrays among others.

5.1 User Side Changes

1. Users are now required to call "initGeNN()" in the model definition function before adding any populations to the neuronal network model.
2. glbscnt is now call glbSpkCnt for consistency with glbSpkEvtCnt.
3. There is no longer a privileged parameter `Epre`. Spike type events are now defined by a code string `spkEvtThreshold` like proper spikes. The only difference ism that Spike type events are specific to a synapse type rather than a neuron type.
4. The function `setSynapseG` has been deprecated. In a `GLOBALG` scenario, the variables of a synapse group are set to the initial values provided in modeldefinition.
5. Due to the split of synaptic models into [weightUpdateModel](#) and [postSynModel](#), the parameter arrays used during model definition need to be carefully split as well so that each side gets the right parameters. For example, previously

```
float myPNKC_p[3]= {  
    0.0,          // 0 - Erev: Reversal potential  
    -20.0,        // 1 - Epre: Presynaptic threshold potential  
    1.0           // 2 - tau_S: decay time constant for S [ms]  
};
```

would define the parameter array of three parameters, `Erev`, `Epre`, and `tau_S` for a synapse of type `NSYNAPSE`. This now needs to be "split" into

```
float *myPNKC_p= NULL;  
float postExpPNKC[2]={  
    1.0,          // 0 - tau_S: decay time constant for S [ms]  
    0.0           // 1 - Erev: Reversal potential  
};
```

i.e. parameters `Erev` and `tau_S` are moved to the post-synaptic model and its parameter array of two parameters. `Epre` is discontinued as a parameter for `NSYNAPSE`. As a consequence the `weightupdate` model of `NSYNAPSE` has no parameters and one can pass `NULL` for the parameter array in `addSynapse←Population`.

The correct parameter lists for all defined neuron and synapse model types are listed in the [User Manual](#).

Note

If the parameters are not redefined appropriately this will lead to uncontrolled behaviour of models and likely to segmentation faults and crashes.

6. Advanced users can now define variables as type "scalar" when introducing new neuron or synapse types. This will at the code generation stage be translated to the model's floating point type (ftype), `float` or `double`. This works for defining variables as well as in all code snippets. Users can also use the expressions "SCALAR_MAX" and "SCALAR_MIN" for "FLT_MIN", "FLT_MAX", "DBL_MIN" and "DBL_MAX", respectively. Corresponding definitions of `scalar`, `SCALAR_MIN` and `SCALAR_MAX` are also available for user-side code whenever the code-generated file `runner.cc` has been included.
7. The example projects have been re-organized so that wrapper scripts of the `generate_run` type are now all located together with the models they run instead of in a common `tools` directory. Generally the structure now is that each example project contains the wrapper script `generate_run` and a `model` subdirectory which contains the model description file and the user side code complete with Makefiles for Unix and Windows operating systems. The generated code will be deposited in the `model` subdirectory in its own `modelname_CODE` folder. Simulation results will always be deposited in a new sub-folder of the main project directory.
8. The `addSynapsePopulation(...)` function has now more mandatory parameters relating to the introduction of separate `weightupdate` models (pre-synaptic models) and postsynaptic models. The correct syntax for the `addSynapsePopulation(...)` can be found with detailed explanations in the [User Manual](#).
9. We have introduced a simple performance profiling method that users can employ to get an overview over the differential use of time by different kernels. To enable the timers in GeNN generated code, one needs to declare

```
networkmodel.setTiming(TRUE);
```

This will make available and operate GPU-side `cudeEvent` based timers whose cumulative value can be found in the double precision variables `neuron_tme`, `synapse_tme` and `learning_tme`. They measure the accumulated time that has been spent calculating the neuron kernel, synapse kernel and learning kernel, respectively. CPU-side timers for the simulation functions are also available and their cumulative values can be obtained through

```
float x= sdkGetTimerValue(&neuron_timer);
float y= sdkGetTimerValue(&synapse_timer);
float z= sdkGetTimerValue(&learning_timer);
```

The [Insect Olfaction Model](#) example shows how these can be used in the user-side code. To enable timing profiling in this example, simply enable it for GeNN:

```
model.setTiming(TRUE);
```

in `MBody1.cc`'s `modelDefinition` function and define the macro `TIMING` in `classol_sim.h`

```
#define TIMING
```

This will have the effect that timing information is output into `OUTNAME_output/OUTNAME.↵ timingprofile`.

5.2 Developer Side Changes

1. `allocateSparseArrays()` has been changed to take the number of connections, `connN`, as an argument rather than expecting it to have been set in the `Connexion` struct before the function is called as was the arrangement previously.
2. For the case of sparse connectivity, there is now a reverse mapping implemented with `revers` index arrays and a `remap` array that points to the original positions of variable values in the forward array. By this mechanism, `revers` lookups from post to pre synaptic indices are possible but value changes in the sparse array values do only need to be done once.

3. SpkEvt code is no longer generated whenever it is not actually used. That is also true on a somewhat finer granularity where variable queues for synapse delays are only maintained if the corresponding variables are used in synaptic code. True spikes on the other hand are always detected in case the user is interested in them.

[Previous](#) | [Top](#) | [Next](#)

Chapter 6

User Manual

6.1 Contents

[Introduction](#)
[Defining a network model](#)
[Neuron models](#)
[Synapse models](#)
[Post-synaptic integration methods](#)
[Variables in GeNN](#)
[Credits](#)

6.2 Introduction

GeNN is a software library for facilitating the simulation of neuronal network models on NVIDIA CUDA enabled GPU hardware. It was designed with computational neuroscience models in mind rather than artificial neural networks. The main philosophy of GeNN is two-fold:

1. GeNN relies heavily on code generation to make it very flexible and to allow adjusting simulation code to the model of interest and the GPU hardware that is detected at compile time.
2. GeNN is lightweight in that it provides code for running models of neuronal networks on GPU hardware but it leaves it to the user to write a final simulation engine. It so allows maximal flexibility to the user who can use any of the provided code but can fully choose, inspect, extend or otherwise modify the generated code. She can also introduce her own optimisations and in particular control the data flow from and to the GPU in any desired granularity.

This manual gives an overview of how to use GeNN for a novice user and tries to lead the user to more expert use later on. With this we jump right in.

[Previous](#) | [Top](#) | [Next](#)

6.3 Defining a network model

A network model is defined by the user by providing the function

```
void modelDefinition(NNmodel &model)
```

in a separate file with name `name.cc`, where `name` is the name of the model network under consideration. In this function, the following tasks must be completed:

1. The name of the model must be defined:

```
model.setName("MyModel");
```

2. Neuron populations (at least one) must be added (see [Defining neuron populations](#)). The user may add as many neuron populations as she wishes. If resources run out, there will not be a warning but GeNN will fail. However, before this breaking point is reached, GeNN will make all necessary efforts in terms of block size optimisation to accommodate the defined models. All populations should have a unique name.
3. Synapse populations (zero or more) can be added (see [Defining synapse populations](#)). Again, the number of synaptic connection populations is unlimited other than by resources.

Note

GeNN uses the convention where C/C++ files end in `.cc`. If this is not adhered to the build script `buildmodel.sh` will not recognise the model definition file.

6.3.1 Defining neuron populations

Neuron populations are added using the function

```
model.addNeuronPopulation(name, n, TYPE, para, ini);
```

where the arguments are:

- `const char* name`: Name of the neuron population
- `int n`: number of neurons in the population
- `int TYPE`: Type of the neurons, refers to either a standard type (see [Neuron models](#)) or user-defined type
- `float *para`: Parameters of this neuron type
- `float *ini`: Initial values for variables of this neuron type

The user may add as many neuron populations as the model necessitates. They should all have unique names. The possible values for the arguments, predefined models and their parameters and initial values are detailed [Neuron models](#) below.

6.3.2 Defining synapse populations

Synapse populations are added with the command

```
model.addSynapsePopulation("name", sType, sConn, gType, delay, postSyn, "preName", "postName", sIni, sParam, postSynIni, postSynParam);
```

where the arguments are

- `const char* name`: The name of the synapse population
- `int sType`: The type of synapse to be added. See [Models](#) below for the available predefined synapse types.
- `int sConn`: The type of synaptic connectivity. the options currently are "ALLTOALL", "DENSE", "SPARSE" (see [Connectivity types](#))

- `int gType`: The way how the synaptic conductivity `g` will be defined. Options are "INDIVIDUALG", "GLOBALG", "INDIVIDUALID". For their meaning, see [Conductance definition methods](#) below.
- `int delay`: Synaptic delay (in multiples of the simulation time step `DT`).
- `int postSyn`: Postsynaptic integration method. See [Post-synaptic integration methods](#) for predefined types.
- `char* preName`: Name of the (existing!) pre-synaptic neuron population.
- `char* postName`: Name of the (existing!) post-synaptic neuron population.
- `float* sIni`: A C-array of floats containing initial values for the (pre-) synaptic variables.
- `float* sParam`: A C-array of floats that contains parameter values (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored. For pre-defined synapse types the required parameters and their meaning are listed in [NSYNAPSE \(No Learning\)](#) below.

Note

If the synapse conductance definition type is "GLOBALG" then the global value of the synapse conductances is taken from the initial value provided in `sINI`. (The function `setSynapseG()` from earlier versions of GeNN has been deprecated).

Synaptic updates can occur per "true" spike (i.e at one point per spike, e.g. after a threshold was crossed) or for all "spike type events" (e.g. all points above a given threshold). This is defined within each given synapse type.

[Previous](#) | [Top](#) | [Next](#)

6.4 Neuron models

There is a number of predefined models which can be chosen in the `addNeuronGroup(...)` function by their unique cardinal number, starting from 0. For convenience, C variables with readable names are predefined

- 0: [MAPNEURON](#)
- 1: [POISSONNEURON](#)
- 2: [TRAUBMILES](#)
- 3: [IZHIKEVICH](#)
- 4: [IZHIKEVICH_V](#)

Note

It is best practice to not depend on the unique cardinal numbers but use predefined names. While it is not intended that the numbers will change the unique names are guaranteed to work in all future versions of GeNN.

6.4.1 MAPNEURON (Map Neurons)

The MAPNEURON type is a map based neuron model based on ? but in the 1-dimensional map form used in ? :

$$V(t + \Delta t) = \begin{cases} V_{\text{spike}} \left(\frac{\alpha V_{\text{spike}}}{V_{\text{spike}} - V(t) \beta I_{\text{syn}}} + y \right) & V(t) \leq 0 \\ V_{\text{spike}} (\alpha + y) & V(t) \leq V_{\text{spike}} (\alpha + y) \text{ \& } V(t - \Delta t) \leq 0 \\ -V_{\text{spike}} & \text{otherwise} \end{cases}$$

Note

The MAPNEURON type only works as intended for the single time step size of $\Delta T = 0.5$.

The MAPNEURON type has 2 variables:

- V - the membrane potential
- preV - the membrane potential at the previous time step

and it has 4 parameters:

- V_{spike} - determines the amplitude of spikes, typically -60mV
- α - determines the shape of the iteration function, typically $\alpha = 3$
- y - "shift / excitation" parameter, also determines the iteration function, originally, $y = -2.468$
- β - roughly speaking equivalent to the input resistance, i.e. it regulates the scale of the input into the neuron, typically $\beta = 2.64 \text{ M}\Omega$.

Note

The initial values array for the MAPNEURON type needs two entries for V and V_{pre} and the parameter array needs four entries for V_{spike} , α , y and β , *in that order*.

6.4.2 POISSONNEURON (Poisson Neurons)

Poisson neurons have constant membrane potential (V_{rest}) unless they are activated randomly to the V_{spike} value if $(t - \text{SpikeTime}) > t_{\text{refract}}$.

It has 3 variables:

- V - Membrane potential
- Seed - Seed for random number generation
- SpikeTime - Time at which the neuron spiked for the last time

and 4 parameters:

- t_{rate} - Firing rate
- t_{refract} - Refractory period
- V_{spike} - Membrane potential at spike (mV)
- V_{rest} - Membrane potential at rest (mV)

Note

The initial values array for the `POISSONNEURON` type needs three entries for `V`, `Seed` and `SpikeTime` and the parameter array needs four entries for `therate`, `trefract`, `Vspike` and `Vrest`, *in that order*. Internally, GeNN uses a linear approximation for the probability of firing a spike in a given time step of size `DT`, i.e. the probability of firing is `therate` times `DT`: $p = \lambda \Delta t$. This approximation is usually very good, especially for typical, quite small time steps and moderate firing rates. However, it is worth noting that the approximation becomes poor for very high firing rates and large time steps. An unrelated problem may occur with very low firing rates and small time steps. In that case it can occur that the firing probability is so small that the granularity of the 64 bit integer based random number generator begins to show. The effect manifests itself in that small changes in the firing rate do not seem to have an effect on the behaviour of the Poisson neurons because the numbers are so small that only if the random number is identical 0 a spike will be triggered.

6.4.3 TRAUBMILES (Hodgkin-Huxley neurons with Traub & Miles algorithm)

This conductance based model has been taken from Traub1991 and can be described by the equations:

$$\begin{aligned} C \frac{dV}{dt} &= -I_{Na} - I_K - I_{leak} - I_M - I_{i,DC} - I_{i,syn} - I_i, \\ I_{Na}(t) &= g_{Na} m_i(t)^3 h_i(t) (V_i(t) - E_{Na}) \\ I_K(t) &= g_K n_i(t)^4 (V_i(t) - E_K) \\ \frac{dy(t)}{dt} &= \alpha_y(V(t))(1 - y(t)) - \beta_y(V(t))y(t), \end{aligned}$$

where $y_i = m, h, n$, and

$$\begin{aligned} \alpha_n &= 0.032(-50 - V) / (\exp((-50 - V)/5) - 1) \\ \beta_n &= 0.5 \exp((-55 - V)/40) \\ \alpha_m &= 0.32(-52 - V) / (\exp((-52 - V)/4) - 1) \\ \beta_m &= 0.28(25 + V) / (\exp((25 + V)/5) - 1) \\ \alpha_h &= 0.128 \exp((-48 - V)/18) \\ \beta_h &= 4 / (\exp((-25 - V)/5) + 1). \end{aligned}$$

and typical parameters are $C = 0.143$ nF, $g_{leak} = 0.02672$ μ S, $E_{leak} = -63.563$ mV, $g_{Na} = 7.15$ μ S, $E_{Na} = 50$ mV, $g_K = 1.43$ μ S, $E_K = -95$ mV.

It has 4 variables:

- V - membrane potential E
- m - probability for Na channel activation m
- h - probability for not Na channel blocking h
- n - probability for K channel activation n

and 7 parameters:

- g_{Na} - Na conductance in $1/(\text{mOhms} * \text{cm}^2)$
- E_{Na} - Na equi potential in mV
- g_K - K conductance in $1/(\text{mOhms} * \text{cm}^2)$
- E_K - K equi potential in mV
- g_l - Leak conductance in $1/(\text{mOhms} * \text{cm}^2)$
- E_l - Leak equi potential in mV
- C_{mem} - Membrane capacity density in $\mu\text{F}/\text{cm}^2$

Note

Internally, the ordinary differential equations defining the model are integrated with a linear Euler algorithm and GeNN integrates 25 internal time steps for each neuron for each network time step. I.e., if the network is simulated at $\Delta T = 0.1$ ms, then the neurons are integrated with a linear Euler algorithm with $\Delta T = 0.004$ ms.

6.4.4 IZHKEVICH (Izhikevich neurons with fixed parameters)

This is the Izhikevich model with fixed parameters. It is usually described as

$$\begin{aligned}\frac{dV}{dt} &= 0.04V^2 + 5V + 140 - U + I, \\ \frac{dU}{dt} &= a(bV - U),\end{aligned}$$

I is an external input current and the voltage V is reset to parameter c and U incremented by parameter d , whenever $V \geq 30$ mV. This is paired with a particular integration procedure of two 0.5 ms Euler time steps for the V equation followed by one 1 ms time step of the U equation. Because of its popularity we provide this model in this form here even though due to the details of the usual implementation it is strictly speaking inconsistent with the displayed equations.

Variables are:

- V - Membrane potential
- U - Membrane recovery variable

Parameters are:

- a - time scale of U
- b - sensitivity of U
- c - after-spike reset value of V
- d - after-spike reset value of U

6.4.5 IZHKEVICH_V (Izhikevich neurons with variable parameters)

This is the same model as [IZHKEVICH \(Izhikevich neurons with fixed parameters\)](#) IZHKEVICH but parameters are defined as "variables" in order to allow users to provide individual values for each individual neuron instead of fixed values for all neurons across the population.

Accordingly, the model has the Variables:

- V - Membrane potential
- U - Membrane recovery variable
- a - time scale of U
- b - sensitivity of U
- c - after-spike reset value of V
- d - after-spike reset value of U

and no parameters.

6.4.6 Defining your own neuron type

In order to define a new neuron type for use in a GeNN application, it is necessary to populate an object of class `neuronModel` and append it to the global vector `nModels`. The `neuronModel` class has several data members that make up the full description of the neuron model:

- `simCode` of type `string`: This needs to be assigned a C++ string that contains the code for executing the integration of the model for one time step. Within this code string, variables need to be referred to by `$(NAME)`, where `NAME` is the name of the variable as defined in the vector `varNames`. The code may refer to the predefined primitives `DT` for the time step size and `Isyn` for the total incoming synaptic current.

Example:

```
neuronModel model;
model.simCode=String("$ (V) +=  (-$(a) $(V) + $(Isyn)) *DT;");
```

would implement a leaky integrator $\frac{dV}{dt} = -aV + I_{syn}$.

- `varNames` of type `vector<string>`: This vector needs to be filled with the names of variables that make up the neuron state. The variables defined here as `NAME` can then be used in the syntax `$(NAME)` in the code string.

Example:

```
model.varNames.push_back(String("V"));
```

would add the variable `V` as needed by the code string in the example above.

- `varTypes` of type `vector<string>`: This vector needs to be filled with the variable type (e.g. "float", "double", etc) for the variables defined in `varNames`. Types and variables are matched to each other by position in the respective vectors.

Example:

```
model.varTypes.push_back(String("float"));
```

would designate the variable `V` to be of type float.

Note

Variable names and variable types are matched by their position in the corresponding arrays `varNames` and `varTypes`, i.e. the 0th entry of `varNames` will have the type stored in the 0th entry of `varTypes` and so on.

- `pNames` of type `vector<string>`: This vector will contain the names of parameters relevant to the model. If defined as `NAME` here, they can then be referenced as `$(NAME)` in the code string. The length of this vector determines the expected number of parameters in the initialisation of the neuron model. Parameters are currently assumed to be always of type float.

```
model.pNames.push_back(String("a"));
```

stores the parameter `a` needed in the code example above.

- `dpNames` of type `vector<string>`: Names of "dependent parameters". Dependent parameters are a mechanism for enhanced efficiency when running neuron models. If parameters with model-side meaning, such as time constants or conductances always appear in a certain combination in the model, then it is more efficient to pre-compute this combination and define it as a dependent parameter. This vector contains the names of such dependent parameters.

For example, if in the example above the original model had been $\frac{dV}{dt} = -g/CV + I_{syn}$. Then one could define the code string and parameters as

```
model.simCode=String("$ (V) +=  (-$(a) $(V) + $(Isyn)) *DT;");
model.varNames.push_back(String("V"));
model.varTypes.push_back(String("float"));
model.pNames.push_back(String("g"));
model.pNames.push_back(String("C"));
model.dpNames.push_back(String("a"));
```

- `dps` of type `dpclass*`: The dependent parameter class, i.e. an implementation of `dpclass` which would return the value for dependent parameters when queried for them. E.g. in the example above it would return `a` when queried for dependent parameter 0 through `dpclass::calculateDerivedParameter()`. Examples how this is done can be found in the pre-defined classes, e.g. `expDecayDp`, `pwSTDP`, `rulkovdp` etc.
- `tmpVarNames` of type `vector<string>`: This vector can be used to request additional variables that are not part of the state of the neuron but used only as temporary storage during evaluation of the integration time step.
For example, one could define

```
model.tmpVarNames.push_back(String("a"));
model.tmpVarNames.push_back(String("float"));
model.simCode= String("$ (a) = $ (g) / $ (C); \n\
$ (V) += - $ (a) * $ (V); \n\");
```

which would be equivalent to

```
model.simCode= String("float $ (a) = $ (g) / $ (C); \n\
$ (V) += - $ (a) * $ (V); \n\");
```

- `tmpVarTypes` of type `vector<string>`: This vector will contain the variable types of the temporary variables, matched up by position as usual.

Once the completed `neuronModel` object is appended to the `nModels` vector,

```
nModels.push_back(model);
```

it can be used in network descriptions by referring to its cardinal number in the `nModels` vector. I.e., if the model is added as the 4th entry, it would be model "3" (counting starts at 0 in usual C convention). The information of the cardinal number of a new model can be obtained by referring to `nModels.size()` right before appending the new model, i.e. a typical use case would be.

```
int myModel= nModels.size();
nModels.push_back(model);
```

. Then one can use the model as

```
networkModel.addNeuronPopulation(..., myModel, ...);
```

6.4.7 Explicit current input to neurons

The user can decide whether a neuron group receives external input current or not in addition to the synaptic input that it receives from the network. External input to a neuron group is activated by calling `activateDirectInput` function. It receives two arguments: The first argument is the name of the neuron group to receive input current. The second parameter defines the type of input. Current options are:

- 0: NOINP: Neuron group receives no input. This is the value by default when the explicit input is not activated.
- 1: CONSTINP: All the neurons receive a constant input value. This value may also be defined as a variable, i.e. be time-dependent.
- 2: MATINP: The input is read from a file containing the input matrix.
- 3: INPRULE: The input is defined as a rule. It differs from CONSTINP in the way that the input is injected: CONSTINP creates a value which is modified in real time, hence complex instructions are limited. INPRULE creates an input array which will be copied into the device memory.

6.5 Synapse models

6.5.1 Models

Currently 3 predefined synapse models are available:

- 0: [NSYNAPSE](#)
- 1: [NGRADSYNAPSE](#)
- 2: [LEARN1SYNAPSE](#)

6.5.2 NSYNAPSE (No Learning)

If this model is selected, no learning rule is applied to the synapse. The model has 3 parameters:

- τ_S - decay time constant for S [ms]
- E_{pre} : Presynaptic threshold potential
- E_{rev} - Reversal potential

(* DEPRECATED: Decay time constant and reversal potential in synapse parameters are not used anymore and they will be removed in the next release. They should be defined in the postsynaptic mechanisms.)

6.5.3 NGRADSYNAPSE (Graded Synapse)

In a graded synapse, the conductance is updated gradually with the rule:

$$g_{Syn} = g * \tanh((E - E_{pre})/V_{slope})$$

The parameters are:

- E_{rev} : Reversal potential
- E_{pre} : Presynaptic threshold potential
- τ_S : Decay time constant for S [ms]
- V_{slope} : Activation slope of graded release

(* DEPRECATED: Decay time constant and reversal potential in synapse parameters are not used anymore and they will be removed in the next release. They should be defined in the postsynaptic mechanisms.)

6.5.4 LEARN1SYNAPSE (Learning Synapse with a Primitive Role)

This is a simple STDP rule including a time delay for the finite transmission speed of the synapse, defined as a piecewise function.

This model has 13 parameters:

- E_{rev} : Reversal potential
- E_{pre} : Presynaptic threshold potential
- τ_S : Decay time constant for S [ms]
- $TLRN$: Time scale of learning changes

- `TCHNG` : Width of learning window
- `TDECAY` : Time scale of synaptic strength decay
- `TPUNISH10` : Time window of suppression in response to 1/0
- `TPUNISH01` : Time window of suppression in response to 0/1
- `GMAX` : Maximal conductance achievable
- `GMID` : Midpoint of sigmoid g filter curve
- `GSLOPE` : Slope of sigmoid g filter curve
- `TAUSHIFT` : Shift of learning curve
- `GSYN0` : Value of syn conductance g decays to

For more details, see ?.

6.5.5 Defining a new synapse model

There is a way to define a synapse model the same way a [neuronModel](#) can be defined. This feature is still experimental and supported only with the SpineML and development branches.

A synapse model is a [weightUpdateModel](#) object which consists of variables, parameters, and three string objects which will be substituted in the code. These three strings are:

- `simCode` : Simulation code that is used when a true spike is detected (only one point after `Vthresh`)
- `simCodeEvt` : Simulation code that is used for spike events (all the instances where `Vm > Vthres`)
- `simLearnPost` : Simulation code which is used in the `learnSynapsesPost` kernel/function, where postsynaptic neuron spikes before the presynaptic neuron in the STDP window. Usually this is simply the conductance update rule defined by the other `simCode` elements above, with negative timing. This code is needed as `simCode` and `simCodeEvt` are used after spanning the presynaptic spikes, where it is not possible to detect where a postsynaptic neuron fired before the presynaptic neuron.

These codes would include update functions for adding up conductances for that neuron model and for changes in conductances for the next time step (learning). Conductance in this code should be referred to as `$("G")`

6.5.6 Connectivity types

If `INDIVIDUALG` is used with `ALLTOALL` connectivity, the conductance values are stored in an array called `"ftype * gp<synapseName>"`, defined and initialised in `runner.cc` in the generated code.

If `INDIVIDUALID` is used with `ALLTOALL` connectivity, the connectivity matrix is stored in an array called `"unsigned int * gp<synapseName>"`

If connectivity is of `SPARSE` type, connectivity and conductance values are stored in a struct in order to occupy minimum memory needed. The struct `Conductance` contains 3 array members and one integer member: 1: `unsigned int connN`: number of connections in the population. This value is needed for allocation of arrays. 2: `floattype * gp`: Values of conductances. The indices that correspond to these values are defined in a pre-to-post basis by the following arrays: 4: `unsigned int gInd`, of size `connN`: Indices of corresponding postsynaptic neurons concatenated for each presynaptic neuron. 3: `unsigned int * gIndInG`, of size `model.neuronN[model.synapseSource[synInd]]+1`: This array defines from which index in the `gp` array the indices in `gInd` would correspond to the presynaptic neuron that corresponds to the index of the `gIndInG` array, with the number of connections being the size of `gInd`. More specifically, `gIndInG[n+1]-gIndInG[n]` would give the number of postsynaptic connections for neuron `n`, and the conductance values are stored in `gp[gIndInG[n]]` to `gp[gIndInG[n]+1]`. if there are no connections for a presynaptic neuron, then `gp[gIndInG[n]]=gp[gIndInG[n]+1]`.

For example, consider a network of two presynaptic neurons connected to three postsynaptic neurons: 0th presynaptic neuron connected to 1st and 2nd postsynaptic neurons, the 1st presynaptic neuron connected to 0th and 2nd neurons. The struct `Conductance` should have these members, with indexing from 0: `ConnN = 4 gp= gPre0-Post1 gpre0-post2 gpre1-post0 gpre1-post2 gInd= 1 2 0 2 gIndIng= 0 2 4`

See `tools/gen_syns_sparse_lzhModel` used in `lzh_sparse` project to see a working example.

6.5.7 Conductance definition methods

The available options work as follows:

- **INDIVIDUALG:** When this option is chosen in the `addSynapsePopulation` command, GeNN reserves an array of size `n_pre x n_post` float for individual conductance values for each combination of pre and postsynaptic neuron. The actual values of the conductances are passed at runtime from the user side code, using the `copyGToDevice` function.
- **GLOBALG:** When this option is chosen, the `addSynapsePopulation` command must be followed within the `modelDefinition` function by a call to `setSynapseG` for this synapse population. This option can only be sensibly combined with connectivity type `ALLTOALL`.
- **INDIVIDUALID:** When this option is chosen, GeNN expects to use the same maximal conductance for all existing synaptic connections but which synapses exist will be defined at runtime from the user side code.

[Previous](#) | [Top](#) | [Next](#)

6.6 Post-synaptic integration methods

The shape of the postsynaptic current can be defined by either using a predefined method or by adding a new `postSynModel` object.

- **EXPDECAY:** Exponential decay. Decay time constant and reversal potential parameters are needed for this postsynaptic mechanism.

[Previous](#) | [Top](#) | [Next](#)

6.7 Variables in GeNN

There are different types of variables that users of GeNN should be aware of and should know how to use.

6.7.1 Neuron and Synapse variables

One type of variables are those that originate from a neuron model type, `weightupdate` model type or post-synaptic model type. The core name of the variable is defined when the model type is introduced, i.e. with a statement such as

```
neuronModel model;
model.varNames.push_back(String"x");
model.varTypes.push_back(String("double"));
...
int myModel= nModels.size();
nModels.push_back(model);
```

This declares that whenever the defined model type of cardinal number `myModel` is used, there will be a variable of core name `x`. The full GeNN name of this variable is obtained by directly concatenating the core name with the name of the neuron population in which the model type has been used, i.e. after a definition

```
networkModel.addNeuronPopulation("EN", n, myModel, ...);
```

there will be a variable `xEN` of type `double*` available in the global namespace of the simulation program. GeNN will pre-allocated this C array to the correct size of elements corresponding to the size of the neuron population, `n` in the example above. GeNN will also free these variables when the provided function `freeMem()` is called. Users can otherwise manipulate these variable arrays as they wish. For convenience, GeNN provides functions `pullXXfromDevice()` and `pushXXtoDevice()` to copy the variables associated to a neuron population `XX` from the device into host memory and vice versa. E.g.

```
pullENfromDevice();
```

would copy the C array `xEN` from device memory into host memory (and any other variables that the neuron type of the population `EN` may have).

The user can also directly use CUDA memory copy commands independent of the provided convenience function. The relevant device pointers for all variables that exist in host memory have the same name prefixed with `d_`. For example, the copy command that would be contained in `pullENfromDevice()` might look like

```
unsigned int size;
size = sizeof(double) * nEN;
cudaMemcpy(xEN, d_xEN, size, cudaMemcpyDeviceToHost);
```

where `n` is an integer containing the population size of the `EN` neuron population.

The same convention as for neuron variables applies for the variables of synapse groups, both for those originating from weightupdate models and from post-synaptic models, e.g. the variables in type `NSYNAPSE` contain the variable `g` of type `float`. Then, after

```
networkModel.addSynapsePopulation("ENIN", NSYNAPSE, ...);
```

there will be a global variable of type `float*` with the name `gENIN` that is pre-allocated to the right size. There will also be a matching device pointer with the name `d_gENIN`.

Note

The content of `gENIN` needs to be interpreted differently for DENSE connectivity and sparse matrix based SPARSE connectivity representations. For DENSE connectivity `gENIN` would contain "`n_pre`" times "`n_post`" elements, ordered along the pre-synaptic neurons as the major dimension, i.e. the value of `gENIN` for the `i`th pre-synaptic neuron and the `j`th post-synaptic neuron would be `gENIN[i*n_post+j]`. The arrangement of values in the SPARSE representation is explained in section [Connectivity types](#)

6.7.2 Pre-defined Variables in GeNN

In addition to the explicitly declared neuron, weightupdate and post-synaptic variables there are some variables that are built-in and modified in a predefined way by GeNN. These variables are used for intermediary calculations and communication between different parts of the generated code. They can be used in the user defined code but no other variables should be defined with these names.

- `addtoInSyn` is a local variable which is used for updating synaptic input. The way it is modified is defined by the weightupdate model, therefore if a user defines her own model she should update this variable to contain the input to the post-synaptic model. At the end of the synaptic update, its value is copied back to the `d_inSyn<synapsePopulation>` variables which will be used in the next step of the neuron update to provide the input to the post-synaptic neurons.

- `Isyn` is a local variable which defines the (summed) input current to a neuron. It is typically the sum of any explicit current input and all synaptic inputs. The way its value is calculated during the update of the postsynaptic neuron is defined by the code provided in the [postSynModel](#). For example, the standard `expDecay` post-synaptic model defines `ps.postSynCurrent= String("$inSyn)*($E-$V)");` which implements a conductance based synapse in which the post-synaptic current is given by $I_{\text{syn}} = g * s * (V_{\text{rev}} - V_{\text{post}})$.

Note

The `addtoinSyn` variables from all incoming synapses are automatically summed and added to the current value of `inSyn`.

The value resulting from the `postSynCurrent` code is assigned to `Isyn` and can then be used in neuron `simCode` like so:

```
$ (V) += ( - $ (V) + $ (Isyn) ) *DT
```

- `sT<neuronName>` is the last spike time in a neuron and is automatically generated for pre- and postsynaptic neuron groups of a synapse group `i` that follows a spike based learning rule (indicated by `usesPostLearning[i]= TRUE` for the `i`th synapse population).

[Previous](#) | [Top](#) | [Next](#)

6.8 Credits

GeNN was created by Thomas Nowotny.

Addition of Izhikevich model, debugging modes and sparse connectivity by Esin Yavuz.

Block size optimisations and delayed synapses by James Turner.

Automatic brackets and dense to sparse network conversion by Alan Diamond.

User-defined synaptic and postsynaptic methods by Alex Cope and Esin Yavuz.

[Previous](#) | [Top](#)

Chapter 7

Hierarchical Index

7.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

classIzh	??
classol	??
CodeHelper	??
CStopWatch	??
dpclass	??
expDecayDp	??
pwSTDP	??
pwSTDP	??
pwSTDP_userdef	??
rulkovdp	??
errTupel	??
histogram	??
inputSpec	??
neuronModel	??
NNmodel	??
postSynModel	??
randomGauss	??
randomGen	??
stdRG	??
stopWatch	??
SynDelay	??
weightUpdateModel	??

Chapter 8

Class Index

8.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

classlzh	..	??
classsol		
	This class contains the methods for running the MBody1 example model	??
CodeHelper		??
CStopWatch		??
dpclass		??
errTupel		??
expDecayDp		
	Class defining the dependent parameter for exponential decay	??
histogram		??
inputSpec		??
neuronModel		
	Class (struct) for specifying a neuron model	??
NNmodel		
	Structure to hold the information that defines synapse dynamics (a model of how synapse variables change over time, independent of or in addition to changes when spikes occur)	??
postSynModel		
	Structure to hold the information that defines a post-synaptic model (a model of how synapses affect post-synaptic neuron variables, classically in the form of a synaptic current). It also allows to define an equation for the dynamics that can be applied to the summed synaptic input variable "insyn"	??
pwSTDP		
	TODO This class definition may be code-generated in a future release	??
pwSTDP_userdef		
	TODO This class definition may be code-generated in a future release	??
randomGauss		
	Class random Gauss encapsulates the methods for generating random neumbers with Gaussian distribution	??
randomGen		
	Class randomGen which implements the ISAAC random number generator for uniformly distributed random numbers	??
rulkovdp		
	Class defining the dependent parameters of teh Rulkov map neuron	??
stdRG		??
stopWatch		??
SynDelay		??

[weightUpdateModel](#)

Structure to hold the information that defines a weightupdate model (a model of how spikes affect synaptic (and/or) (mostly) post-synaptic neuron variables. It also allows to define changes in response to post-synaptic spikes/spike-like events ??

Chapter 9

File Index

9.1 File List

Here is a list of all files with brief descriptions:

lib/include/ensureFtype.h	??
lib/include/extra_neurons.h	??
lib/include/extra_postsynapses.h	??
lib/include/extra_weightupdates.h	??
lib/include/global.h Global header file containing a few global variables. Part of the code generation section	??
lib/include/hr_time.cpp This file contains the implementation of the CStopWatch class that provides a simple timing tool based on the system clock	??
lib/include/hr_time.h This header file contains the definition of the CStopWatch class that implements a simple timing tool using the system clock	??
lib/include/modelSpec.cc	??
lib/include/modelSpec.h Header file that contains the class (struct) definition of neuronModel for defining a neuron model and the class definition of NNmodel for defining a neuronal network model. Part of the code generation and generated code sections	??
lib/include/sparseUtils.cc	??
lib/include/testi.cc	??
lib/include/toString.h Contains a template function for string conversion from const char* to C++ string	??
lib/include/usertools.h	??
lib/include/utils.h This file contains standard utility functions provide within the NVIDIA CUDA software development toolkit (SDK). The remainder of the file contains a function that defines the standard neuron models	??
lib/include/numlib/gauss.cc Contains the implementation of the Gaussian random number generator class randomGauss	??
lib/include/numlib/gauss.h Random number generator for Gaussian random variable with mean 0 and standard deviation 1	??
lib/include/numlib/randomGen.cc Contains the implementation of the ISAAC random number generator class for uniformly distributed random numbers and for a standard random number generator based on the C function rand()	??
lib/include/numlib/randomGen.h Header file containing the class definition for a uniform random generator based on the ISAAC random number generator	??

lib/include/numlib/ simpleBit.h	
Contains three macros that allow simple bit manipulations on an (presumably unsigned) 64 bit integer	??
lib/src/ CodeHelper.cc	??
lib/src/ generateALL.cc	
Main file combining the code for code generation. Part of the code generation section	??
lib/src/ generateCPU.cc	
Functions for generating code that will run the neuron and synapse simulations on the CPU. Part of the code generation section	??
lib/src/ generateKernels.cc	
Contains functions that generate code for CUDA kernels. Part of the code generation section	??
lib/src/ generateRunner.cc	
Contains functions to generate code for running the simulation on the GPU, and for I/O convenience functions between GPU and CPU space. Part of the code generation section	??
SpineML_to_GeNN/ extra_postsynapses.h	??
tmp/ generate_run.cc	??
tmp/ test.cc	??
tmp/model/ classol_sim.cu	??
tmp/model/ classol_sim.h	??
tmp/model/ map_classol.cc	??
tmp/model/ map_classol.h	??
tmp/model/ MBody_userdef.cc	??
userproject/ GeNNHelperKrnls.cu	??
userproject/HHVclampGA_project/ generate_run.cc	??
userproject/HHVclampGA_project/model/ GA.cc	??
userproject/HHVclampGA_project/model/ helper.h	??
userproject/HHVclampGA_project/model/ HHVClamp.cc	
This file contains the model definition of HHVClamp model. It is used in both the GeNN code generation and the user side simulation code. The HHVClamp model implements a population of unconnected Hodgkin-Huxley neurons that evolve to mimick a model run on the CPU, using genetic algorithm techniques	??
userproject/HHVclampGA_project/model/ MBody1.cc	??
userproject/HHVclampGA_project/model/ VClampGA.cu	
Main entry point for the GeNN project demonstrating realtime fitting of a neuron with a GA running mostly on the GPU	??
userproject/HHVclampGA_project/model/ VClampGA.h	??
userproject/include/ HHVClampParameters.h	??
userproject/include/ sizes.h	??
userproject/lzh_sparse_project/ generate_run.cc	??
userproject/lzh_sparse_project/model/ lzh_sim_sparse.cu	??
userproject/lzh_sparse_project/model/ lzh_sparse.cc	??
userproject/lzh_sparse_project/model/ lzh_sparse_model.cc	??
userproject/lzh_sparse_project/model/ lzh_sparse_model.h	??
userproject/lzh_sparse_project/model/ lzh_sparse_sim.h	??
userproject/MBody1_project/ generate_run.cc	??
userproject/MBody1_project/model/ classol_sim.cu	??
userproject/MBody1_project/model/ classol_sim.h	??
userproject/MBody1_project/model/ classol_sim_detailed_output.cu	??
userproject/MBody1_project/model/ map_classol.cc	??
userproject/MBody1_project/model/ map_classol.h	??
userproject/MBody1_project/model/ MBody1.cc	??
userproject/MBody1_project/modelcmp/ classol_sim.cu	??
userproject/MBody1_project/modelcmp/ classol_sim.h	??
userproject/MBody1_project/modelcmp/ map_classol.cc	??
userproject/MBody1_project/modelcmp/ map_classol.h	??
userproject/MBody1_project/modelcmp/ MBody1.cc	??
userproject/MBody_userdef_project/ generate_run.cc	??
userproject/MBody_userdef_project/model/ classol_sim.cu	??

userproject/MBody_userdef_project/model/classol_sim.h	??
userproject/MBody_userdef_project/model/map_classol.cc	??
userproject/MBody_userdef_project/model/map_classol.h	??
userproject/MBody_userdef_project/model/MBody_userdef.cc	??
userproject/OneComp_project/generate_run.cc	??
userproject/OneComp_project/model/OneComp.cc	??
userproject/OneComp_project/model/OneComp_model.cc	??
userproject/OneComp_project/model/OneComp_model.h	??
userproject/OneComp_project/model/OneComp_sim.cu	??
userproject/OneComp_project/model/OneComp_sim.h	??
userproject/PoissonIzh_project/generate_run.cc	??
userproject/PoissonIzh_project/model/PoissonIzh-model.cc	??
userproject/PoissonIzh_project/model/PoissonIzh-model.h	??
userproject/PoissonIzh_project/model/PoissonIzh.cc	??
userproject/PoissonIzh_project/model/PoissonIzh_sim.cu	??
userproject/PoissonIzh_project/model/PoissonIzh_sim.h	??
userproject/SynDelay_project/SynDelay.cc	??
userproject/SynDelay_project/SynDelaySim.cu	??
userproject/SynDelay_project/SynDelaySim.h	??
userproject/tools/gen_input_structured.cc	??
userproject/tools/gen_kcdn_syns.cc	
This file is part of a tool chain for running the classol/MBody1 example model	??
userproject/tools/gen_pnkc_syns.cc	
This file is part of a tool chain for running the classol/MBody1 example model	??
userproject/tools/gen_pnkc_syns_indivID.cc	
This file is part of a tool chain for running the classol/MBody1 example model	??
userproject/tools/gen_pnlhi_syns.cc	
This file is part of a tool chain for running the classol/MBody1 example model	??
userproject/tools/gen_syns_sparse.cc	
This file generates the arrays needed for sparse connectivity. The connectivity is saved to a file for each variable and can then be read to fill the struct of connectivity	??
userproject/tools/gen_syns_sparse_izhModel.cc	
This file is part of a tool chain for running the Izhikevich network model	??

Chapter 10

Class Documentation

10.1 classlzh Class Reference

```
#include <Izh_sparse_model.h>
```

Public Member Functions

- [classlzh](#) ()
- [~classlzh](#) ()
- void [init](#) (unsigned int)
- void [allocate_device_mem_patterns](#) ()
- void [allocate_device_mem_input](#) ()
- void [copy_device_mem_input](#) ()
- void [read_sparsesyns_par](#) (int, struct Conductance, FILE *, FILE *, FILE *, float *)
- void [gen_alltoall_syns](#) (float *, unsigned int, unsigned int, float)
- void [free_device_mem](#) ()
- void [write_input_to_file](#) (FILE *)
- void [read_input_values](#) (FILE *)
- void [create_input_values](#) ()
- void [run](#) (float, unsigned int)
- void [getSpikesFromGPU](#) ()
Method for copying all spikes of the last time step from the GPU.
- void [getSpikeNumbersFromGPU](#) ()
Method for copying the number of spikes in all neuron populations that have occurred during the last time step.
- void [output_state](#) (FILE *, unsigned int)
- void [output_spikes](#) (FILE *, unsigned int)
- void [output_params](#) (FILE *, FILE *)
- void [sum_spikes](#) ()
- void [setInput](#) (unsigned int)
- void [randomizeVar](#) (float *, float, unsigned int)
- void [randomizeVarSq](#) (float *, float, unsigned int)
- void [initializeAllVars](#) (unsigned int)

Public Attributes

- [NNmodel](#) model
- float * [input1](#)
- float * [input2](#)

- float * [d_input1](#)
- float * [d_input2](#)
- unsigned int [sumPExc](#)
- unsigned int [sumPlnh](#)

10.1.1 Constructor & Destructor Documentation

10.1.1.1 `classlzh::classlzh ()`

10.1.1.2 `classlzh::~~classlzh ()`

10.1.2 Member Function Documentation

10.1.2.1 `void classlzh::allocate_device_mem_input ()`

10.1.2.2 `void classlzh::allocate_device_mem_patterns ()`

10.1.2.3 `void classlzh::copy_device_mem_input ()`

10.1.2.4 `void classlzh::create_input_values ()`

10.1.2.5 `void classlzh::free_device_mem ()`

10.1.2.6 `void classlzh::gen_alltoall_syns (float * g, unsigned int nPre, unsigned int nPost, float gscale)`

Parameters

<i>gscale</i>	Generate random conductivity values for an all to all network
---------------	---

10.1.2.7 `void classlzh::getSpikeNumbersFromGPU ()`

Method for copying the number of spikes in all neuron populations that have occurred during the last time step. This method is a simple wrapper for the convenience function `copySpikeNFromDevice()` provided by GeNN.

10.1.2.8 `void classlzh::getSpikesFromGPU ()`

Method for copying all spikes of the last time step from the GPU. This is a simple wrapper for the convenience function `copySpikesFromDevice()` which is provided by GeNN.

10.1.2.9 `void classlzh::init (unsigned int which)`

10.1.2.10 `void classlzh::initializeAllVars (unsigned int which)`

10.1.2.11 `void classlzh::output_params (FILE * f, FILE * f2)`

10.1.2.12 `void classlzh::output_spikes (FILE * f, unsigned int which)`

10.1.2.13 `void classlzh::output_state (FILE * f, unsigned int which)`

10.1.2.14 `void classlzh::randomizeVar (float * Var, float strength, unsigned int neuronGrp)`

10.1.2.15 `void classlzh::randomizeVarSq (float * Var, float strength, unsigned int neuronGrp)`

10.1.2.16 void classlzh::read_input_values (FILE *)

10.1.2.17 void classlzh::read_sparsesyns_par (int *synInd*, struct Conductance *C*, FILE * *f_ind*, FILE * *f_indInG*, FILE * *f_g*, float * *g*)

Parameters

<i>g</i>	File handle for a file containing sparse conductivity values
----------	--

10.1.2.18 void classlzh::run (float *runtime*, unsigned int *which*)

10.1.2.19 void classlzh::setInput (unsigned int *which*)

10.1.2.20 void classlzh::sum_spikes ()

10.1.2.21 void classlzh::write_input_to_file (FILE * *f*)

10.1.3 Member Data Documentation

10.1.3.1 float* classlzh::d_input1

10.1.3.2 float * classlzh::d_input2

10.1.3.3 float* classlzh::input1

10.1.3.4 float * classlzh::input2

10.1.3.5 NNmodel classlzh::model

10.1.3.6 unsigned int classlzh::sumPExc

10.1.3.7 unsigned int classlzh::sumPlnh

The documentation for this class was generated from the following files:

- [userproject/lzh_sparse_project/model/lzh_sparse_model.h](#)
- [userproject/lzh_sparse_project/model/lzh_sparse_model.cc](#)

10.2 classol Class Reference

This class contains the methods for running the MBody1 example model.

```
#include <map_classol.h>
```

Public Member Functions

- [classol](#) ()
- [~classol](#) ()
- void [init](#) (unsigned int)
Method for initialising variables.
- void [allocate_device_mem_patterns](#) ()
Method for allocating memory on the GPU device to hold the input patterns.
- void [free_device_mem](#) ()

- Methods for unallocating the memory for input patterns on the GPU device.*

 - void [read_pnkcsyns](#) (FILE *)

Method for reading the connectivity between PNs and KCs from a file.

 - void [read_sparsesyns_par](#) (int, struct Conductance, FILE *, FILE *, FILE *)
 - void [write_pnkcsyns](#) (FILE *)

Method for writing the connectivity between PNs and KCs back into file.

 - void [read_pnlhisyns](#) (FILE *)

Method for reading the connectivity between PNs and LHIs from a file.

 - void [write_pnlhisyns](#) (FILE *)

Method for writing the connectivity between PNs and LHIs to a file.

 - void [read_kcdnsyns](#) (FILE *)

Method for reading the connectivity between KCs and DNs (detector neurons) from a file.

 - void [write_kcdnsyns](#) (FILE *)

Method to write the connectivity between KCs and DNs (detector neurons) to a file.

 - void [read_input_patterns](#) (FILE *)

Method for reading the input patterns from a file.

 - void [generate_baserates](#) ()

Method for calculating the baseline rates of the Poisson input neurons.

 - void [run](#) (float, unsigned int)

Method for simulating the model for a given period of time.

 - void [output_state](#) (FILE *, unsigned int)

Method for copying from device and writing out to file of the entire state of the model.

 - void [getSpikesFromGPU](#) ()

Method for copying all spikes of the last time step from the GPU.

 - void [getSpikeNumbersFromGPU](#) ()

Method for copying the number of spikes in all neuron populations that have occurred during the last time step.

 - void [output_spikes](#) (FILE *, unsigned int)

Method for writing the spikes occurred in the last time step to a file.

 - void [sum_spikes](#) ()

Method for summing up spike numbers.

 - void [get_kcdnsyns](#) ()

Method for copying the synaptic conductances of the learning synapses between KCs and DNs (detector neurons) back to the CPU memory.

 - [classol](#) ()
 - [~classol](#) ()
 - void [init](#) (unsigned int)
 - void [allocate_device_mem_patterns](#) ()
 - void [free_device_mem](#) ()
 - void [read_pnkcsyns](#) (FILE *)
 - void [read_sparsesyns_par](#) (int, struct Conductance, [scalar](#) *, FILE *, FILE *, FILE *)
 - void [write_pnkcsyns](#) (FILE *)
 - void [read_pnlhisyns](#) (FILE *)
 - void [write_pnlhisyns](#) (FILE *)
 - void [read_kcdnsyns](#) (FILE *)
 - void [write_kcdnsyns](#) (FILE *)
 - void [read_input_patterns](#) (FILE *)
 - void [generate_baserates](#) ()
 - void [run](#) ([scalar](#), unsigned int)

Method for simulating the model for a given period of time.

 - void [output_state](#) (FILE *, unsigned int)
 - void [getSpikesFromGPU](#) ()
 - void [getSpikeNumbersFromGPU](#) ()

- void [output_spikes](#) (FILE *, unsigned int)
- void [sum_spikes](#) ()
- void [get_kcdnsyns](#) ()
- [classol](#) ()
- [~classol](#) ()
- void [init](#) (unsigned int)
- void [allocate_device_mem_patterns](#) ()
- void [free_device_mem](#) ()
- void [read_pnkcsyns](#) (FILE *)
- void [read_sparsesyns_par](#) (int, struct Conductance, FILE *, FILE *, FILE *)
- void [write_pnkcsyns](#) (FILE *)
- void [read_pnlhisyns](#) (FILE *)
- void [write_pnlhisyns](#) (FILE *)
- void [read_kcdnsyns](#) (FILE *)
- void [write_kcdnsyns](#) (FILE *)
- void [read_input_patterns](#) (FILE *)
- void [generate_baserates](#) ()
- void [run](#) (float, unsigned int)
- void [output_state](#) (FILE *, unsigned int)
- void [getSpikesFromGPU](#) ()
- void [getSpikeNumbersFromGPU](#) ()
- void [output_spikes](#) (FILE *, unsigned int)
- void [sum_spikes](#) ()
- void [get_kcdnsyns](#) ()
- [classol](#) ()
- [~classol](#) ()
- void [init](#) (unsigned int)
- void [allocate_device_mem_patterns](#) ()
- void [free_device_mem](#) ()
- void [read_pnkcsyns](#) (FILE *)
- template<class DATATYPE >
void [read_sparsesyns_par](#) (DATATYPE *, int, struct Conductance, FILE *, FILE *, FILE *)
- void [write_pnkcsyns](#) (FILE *)
- void [read_pnlhisyns](#) (FILE *)
- void [write_pnlhisyns](#) (FILE *)
- void [read_kcdnsyns](#) (FILE *)
- void [write_kcdnsyns](#) (FILE *)
- void [read_input_patterns](#) (FILE *)
- void [generate_baserates](#) ()
- void [run](#) (float, unsigned int)
- void [output_state](#) (FILE *, unsigned int)
- void [getSpikesFromGPU](#) ()
- void [getSpikeNumbersFromGPU](#) ()
- void [output_spikes](#) (FILE *, unsigned int)
- void [sum_spikes](#) ()
- void [get_kcdnsyns](#) ()
- [classol](#) ()
- [~classol](#) ()
- void [init](#) (unsigned int)
- void [allocate_device_mem_patterns](#) ()
- void [allocate_device_mem_input](#) ()
- void [copy_device_mem_input](#) ()
- void [free_device_mem](#) ()
- void [write_input_to_file](#) (FILE *)
- void [read_input_values](#) (FILE *)

- void [create_input_values](#) (float t)
- void [run](#) (float, unsigned int)
- void [getSpikesFromGPU](#) ()
- void [getSpikeNumbersFromGPU](#) ()
- void [output_state](#) (FILE *, unsigned int)
- void [output_spikes](#) (FILE *, unsigned int)
- void [sum_spikes](#) ()
- [classol](#) ()
- [~classol](#) ()
- void [init](#) (unsigned int)
- void [allocate_device_mem_patterns](#) ()
- void [allocate_device_mem_input](#) ()
- void [free_device_mem](#) ()
- void [read_PNlzh1syms](#) (float *, FILE *)
- void [read_sparsesyms_par](#) (int, struct Conductance, FILE *, FILE *, FILE *)
- void [read_input_patterns](#) (FILE *)
- void [generate_baserates](#) ()
- void [run](#) (float, unsigned int)
- void [output_state](#) (FILE *, unsigned int)
- void [getSpikesFromGPU](#) ()
- void [getSpikeNumbersFromGPU](#) ()
- void [output_spikes](#) (FILE *, unsigned int)
- void [sum_spikes](#) ()

Public Attributes

- [NNmodel](#) model
- unsigned int * [theRates](#)
- unsigned int * [pattern](#)
- unsigned int * [baserates](#)
- unsigned int * [d_pattern](#)
- unsigned int * [d_baserates](#)
- unsigned int [sumPN](#)
- unsigned int [sumKC](#)
- unsigned int [sumLHI](#)
- unsigned int [sumDN](#)
- unsigned int [size_g](#)
- unsigned int [offset](#)
- uint64_t * [theRates](#)
- scalar * [p_pattern](#)
- uint64_t * [pattern](#)
- uint64_t * [baserates](#)
- uint64_t * [d_pattern](#)
- uint64_t * [d_baserates](#)
- float * [p_pattern](#)
- float * [input1](#)
- float * [d_input1](#)
- unsigned int [sumlzh1](#)

10.2.1 Detailed Description

This class contains the methods for running the MBody1 example model.

10.2.2 Constructor & Destructor Documentation

10.2.2.1 `classol::classol ()`

10.2.2.2 `classol::~~classol ()`

10.2.2.3 `classol::classol ()`

10.2.2.4 `classol::~~classol ()`

10.2.2.5 `classol::classol ()`

10.2.2.6 `classol::~~classol ()`

10.2.2.7 `classol::classol ()`

10.2.2.8 `classol::~~classol ()`

10.2.2.9 `classol::classol ()`

10.2.2.10 `classol::~~classol ()`

10.2.2.11 `classol::classol ()`

10.2.2.12 `classol::~~classol ()`

10.2.3 Member Function Documentation

10.2.3.1 `void classol::allocate_device_mem_input ()`

10.2.3.2 `void classol::allocate_device_mem_input ()`

10.2.3.3 `void classol::allocate_device_mem_patterns ()`

10.2.3.4 `void classol::allocate_device_mem_patterns ()`

10.2.3.5 `void classol::allocate_device_mem_patterns ()`

Method for allocating memory on the GPU device to hold the input patterns.

10.2.3.6 `void classol::allocate_device_mem_patterns ()`

10.2.3.7 `void classol::allocate_device_mem_patterns ()`

10.2.3.8 `void classol::allocate_device_mem_patterns ()`

10.2.3.9 `void classol::copy_device_mem_input ()`

10.2.3.10 `void classol::create_input_values (float t)`

10.2.3.11 `void classol::free_device_mem ()`

10.2.3.12 `void classol::free_device_mem ()`

10.2.3.13 void classol::free_device_mem ()

Methods for unallocating the memory for input patterns on the GPU device.

10.2.3.14 void classol::free_device_mem ()

10.2.3.15 void classol::free_device_mem ()

10.2.3.16 void classol::free_device_mem ()

10.2.3.17 void classol::generate_baserates ()

10.2.3.18 void classol::generate_baserates ()

Method for calculating the baseline rates of the Poisson input neurons.

10.2.3.19 void classol::generate_baserates ()

10.2.3.20 void classol::generate_baserates ()

10.2.3.21 void classol::generate_baserates ()

10.2.3.22 void classol::get_kcdnsyns ()

Method for copying the synaptic conductances of the learning synapses between KCs and DNs (detector neurons) back to the CPU memory.

10.2.3.23 void classol::get_kcdnsyns ()

10.2.3.24 void classol::get_kcdnsyns ()

10.2.3.25 void classol::get_kcdnsyns ()

10.2.3.26 void classol::getSpikeNumbersFromGPU ()

10.2.3.27 void classol::getSpikeNumbersFromGPU ()

10.2.3.28 void classol::getSpikeNumbersFromGPU ()

Method for copying the number of spikes in all neuron populations that have occurred during the last time step.

This method is a simple wrapper for the convenience function copySpikeNFromDevice() provided by GeNN.

10.2.3.29 void classol::getSpikeNumbersFromGPU ()

10.2.3.30 void classol::getSpikeNumbersFromGPU ()

10.2.3.31 void classol::getSpikeNumbersFromGPU ()

10.2.3.32 void classol::getSpikesFromGPU ()

10.2.3.33 void classol::getSpikesFromGPU ()

10.2.3.34 void classol::getSpikesFromGPU ()

Method for copying all spikes of the last time step from the GPU.

This is a simple wrapper for the convenience function copySpikesFromDevice() which is provided by GeNN.

10.2.3.35 void classol::getSpikesFromGPU ()

10.2.3.36 void classol::getSpikesFromGPU ()

10.2.3.37 void classol::getSpikesFromGPU ()

10.2.3.38 void classol::init (unsigned int)

10.2.3.39 void classol::init (unsigned int)

10.2.3.40 void classol::init (unsigned int *which*)

Method for initialising variables.

Parameters

<i>which</i>	Flag defining whether GPU or CPU only version is run
--------------	--

10.2.3.41 void classol::init (unsigned int)

10.2.3.42 void classol::init (unsigned int)

10.2.3.43 void classol::init (unsigned int)

10.2.3.44 void classol::output_spikes (FILE *, unsigned int)

10.2.3.45 void classol::output_spikes (FILE *, unsigned int)

10.2.3.46 void classol::output_spikes (FILE * *f*, unsigned int *which*)

Method for writing the spikes occurred in the last time step to a file.

Parameters

<i>f</i>	File handle for a file to write spike times to
<i>which</i>	Flag determining whether using GPU or CPU only

10.2.3.47 void classol::output_spikes (FILE *, unsigned int)

10.2.3.48 void classol::output_spikes (FILE *, unsigned int)

10.2.3.49 void classol::output_spikes (FILE *, unsigned int)

10.2.3.50 void classol::output_state (FILE *, unsigned int)

10.2.3.51 void classol::output_state (FILE *, unsigned int)

10.2.3.52 void classol::output_state (FILE * *f*, unsigned int *which*)

Method for copying from device and writing out to file of the entire state of the model.

Parameters

<i>f</i>	File handle for a file to write the model state to
<i>which</i>	Flag determining whether using GPU or CPU only

10.2.3.53 void classol::output_state (FILE * , unsigned int)

10.2.3.54 void classol::output_state (FILE * , unsigned int)

10.2.3.55 void classol::output_state (FILE * , unsigned int)

10.2.3.56 void classol::read_input_patterns (FILE *)

10.2.3.57 void classol::read_input_patterns (FILE * f)

Method for reading the input patterns from a file.

Parameters

<i>f</i>	File handle for a file containing input patterns
----------	--

10.2.3.58 void classol::read_input_patterns (FILE *)

10.2.3.59 void classol::read_input_patterns (FILE *)

10.2.3.60 void classol::read_input_patterns (FILE *)

10.2.3.61 void classol::read_input_values (FILE * f)

10.2.3.62 void classol::read_kcdnsyns (FILE * f)

Method for reading the connectivity between KCs and DNs (detector neurons) from a file.

Parameters

<i>f</i>	File handle for a file containing KC to DN (detector neuron) conductivity values
----------	--

10.2.3.63 void classol::read_kcdnsyns (FILE *)

10.2.3.64 void classol::read_kcdnsyns (FILE *)

10.2.3.65 void classol::read_kcdnsyns (FILE *)

10.2.3.66 void classol::read_PNlzh1syns (float * gp, FILE * f)

10.2.3.67 void classol::read_pnkcsyns (FILE * f)

Method for reading the connectivity between PNs and KCs from a file.

Parameters

<i>f</i>	File handle for a file containing PN to KC conductivity values
----------	--

10.2.3.68 void classol::read_pnkcsyns (FILE *)

10.2.3.69 void classol::read_pnkcsyns (FILE *)

10.2.3.70 void classol::read_pnkcsyns (FILE *)

10.2.3.71 void classol::read_pnlhsyns (FILE * *f*)

Method for reading the connectivity between PNs and LHIs from a file.

Parameters

<i>f</i>	File handle for a file containing PN to LHI conductivity values
----------	---

10.2.3.72 void classol::read_pnlhsyns (FILE *)

10.2.3.73 void classol::read_pnlhsyns (FILE *)

10.2.3.74 void classol::read_pnlhsyns (FILE *)

10.2.3.75 void classol::read_sparsesyns_par (int , struct Conductance , FILE * , FILE * , FILE *)

10.2.3.76 void classol::read_sparsesyns_par (int *synInd*, struct Conductance *C*, FILE * *f_ind*, FILE * *f_indInG*, FILE * *f_g*)

Parameters

<i>f_g</i>	File handle for a file containing sparse conductivity values
------------	--

10.2.3.77 void classol::read_sparsesyns_par (int , struct Conductance , FILE * , FILE * , FILE *)

10.2.3.78 void classol::read_sparsesyns_par (int *synInd*, struct Conductance *C*, scalar * *g*, FILE * *f_ind*, FILE * *f_indInG*, FILE * *f_g*)

Parameters

<i>f_g</i>	File handle for a file containing sparse conductivity values
------------	--

10.2.3.79 template<class DATATYPE > void classol::read_sparsesyns_par (DATATYPE * *wuvar*, int *synInd*, struct Conductance *C*, FILE * *f_ind*, FILE * *f_indInG*, FILE * *f_g*)

Parameters

<i>f_g</i>	File handle for a file containing sparse conductivity values
------------	--

10.2.3.80 void classol::run (float , unsigned int)

10.2.3.81 void classol::run (float , unsigned int)

10.2.3.82 void classol::run (float *runtime*, unsigned int *which*)

Method for simulating the model for a given period of time.

Parameters

<i>runtime</i>	Duration of time to run the model for
<i>which</i>	Flag determining whether to run on GPU or CPU only

10.2.3.83 `void classol::run (float , unsigned int)`

10.2.3.84 `void classol::run (scalar runtime, unsigned int which)`

Method for simulating the model for a given period of time.

Parameters

<i>runtime</i>	Duration of time to run the model for
<i>which</i>	Flag determining whether to run on GPU or CPU only

10.2.3.85 `void classol::run (float , unsigned int)`

10.2.3.86 `void classol::sum_spikes ()`

10.2.3.87 `void classol::sum_spikes ()`

10.2.3.88 `void classol::sum_spikes ()`

Method for summing up spike numbers.

10.2.3.89 `void classol::sum_spikes ()`

10.2.3.90 `void classol::sum_spikes ()`

10.2.3.91 `void classol::sum_spikes ()`

10.2.3.92 `void classol::write_input_to_file (FILE * f)`

10.2.3.93 `void classol::write_kcdnsyns (FILE * f)`

Method to write the connectivity between KCs and DNs (detector neurons) to a file.

Parameters

<i>f</i>	File handle for a file to write KC to DN (detectore neuron) conductivity values to
----------	--

10.2.3.94 `void classol::write_kcdnsyns (FILE *)`

10.2.3.95 `void classol::write_kcdnsyns (FILE *)`

10.2.3.96 `void classol::write_kcdnsyns (FILE *)`

10.2.3.97 `void classol::write_pnkcsyns (FILE * f)`

Method for writing the conenctivity between PNs and KCs back into file.

Parameters

<i>f</i>	File handle for a file to write PN to KC conductivity values to
----------	---

10.2.3.98 void classol::write_pnkcsyns (FILE *)

10.2.3.99 void classol::write_pnkcsyns (FILE *)

10.2.3.100 void classol::write_pnkcsyns (FILE *)

10.2.3.101 void classol::write_pnlhisyns (FILE * *f*)

Method for writing the connectivity between PNs and LHIs to a file.

Parameters

<i>f</i>	File handle for a file to write PN to LHI conductivity values to
----------	--

10.2.3.102 void classol::write_pnlhisyns (FILE *)

10.2.3.103 void classol::write_pnlhisyns (FILE *)

10.2.3.104 void classol::write_pnlhisyns (FILE *)

10.2.4 Member Data Documentation

10.2.4.1 unsigned int * classol::baserates

10.2.4.2 uint64_t* classol::baserates

10.2.4.3 unsigned int * classol::d_baserates

10.2.4.4 uint64_t* classol::d_baserates

10.2.4.5 float* classol::d_input1

10.2.4.6 unsigned int * classol::d_pattern

10.2.4.7 uint64_t* classol::d_pattern

10.2.4.8 float* classol::input1

10.2.4.9 NNmodel classol::model

10.2.4.10 unsigned int classol::offset

10.2.4.11 float* classol::p_pattern

10.2.4.12 float * classol::p_pattern

10.2.4.13 unsigned int * classol::pattern

10.2.4.14 uint64_t* classol::pattern

10.2.4.15 unsigned int classol::size_g

10.2.4.16 unsigned int classol::sumDN

10.2.4.17 unsigned int classol::sumIzh1

10.2.4.18 unsigned int classol::sumKC

10.2.4.19 unsigned int classol::sumLHI

10.2.4.20 unsigned int classol::sumPN

10.2.4.21 unsigned int * classol::theRates

10.2.4.22 uint64_t* classol::theRates

The documentation for this class was generated from the following files:

- tmp/model/[map_classol.h](#)
- userproject/OneComp_project/model/[OneComp_model.h](#)
- userproject/PoissonIzh_project/model/[PoissonIzh-model.h](#)
- tmp/model/[map_classol.cc](#)
- userproject/OneComp_project/model/[OneComp_model.cc](#)
- userproject/PoissonIzh_project/model/[PoissonIzh-model.cc](#)

10.3 CodeHelper Class Reference

Public Member Functions

- [CodeHelper](#) ()
- void [setVerbose](#) (bool isVerbose)
- string [openBrace](#) (unsigned int level)
- string [closeBrace](#) (unsigned int level)
- string [endl](#) ()

Public Attributes

- vector< unsigned int > [braces](#)
- bool [verbose](#)

10.3.1 Constructor & Destructor Documentation

10.3.1.1 [CodeHelper::CodeHelper](#) () [[inline](#)]

10.3.2 Member Function Documentation

10.3.2.1 string [CodeHelper::closeBrace](#) (unsigned int *level*) [[inline](#)]

10.3.2.2 string [CodeHelper::endl](#) () [[inline](#)]

10.3.2.3 string [CodeHelper::openBrace](#) (unsigned int *level*) [[inline](#)]

10.3.2.4 void [CodeHelper::setVerbose](#) (bool *isVerbose*) [[inline](#)]

10.3.3 Member Data Documentation

10.3.3.1 `vector<unsigned int> CodeHelper::braces`

10.3.3.2 `bool CodeHelper::verbose`

The documentation for this class was generated from the following file:

- [lib/src/CodeHelper.cc](#)

10.4 CStopWatch Class Reference

```
#include <hr_time.h>
```

Public Member Functions

- [CStopWatch \(\)](#)
- void [startTimer \(\)](#)
This method starts the timer.
- void [stopTimer \(\)](#)
This method stops the timer.
- double [getElapsedTime \(\)](#)
This method returns the time elapsed between start and stop of the timer in seconds.

10.4.1 Constructor & Destructor Documentation

10.4.1.1 `CStopWatch::CStopWatch () [inline]`

10.4.2 Member Function Documentation

10.4.2.1 `double CStopWatch::getElapsedTime ()`

This method returns the time elapsed between start and stop of the timer in seconds.

10.4.2.2 `void CStopWatch::startTimer ()`

This method starts the timer.

10.4.2.3 `void CStopWatch::stopTimer ()`

This method stops the timer.

The documentation for this class was generated from the following files:

- [lib/include/hr_time.h](#)
- [lib/include/hr_time.cpp](#)

10.5 dpclass Class Reference

```
#include <modelSpec.h>
```

Inheritance diagram for dpclass:

Public Member Functions

- [dpclass](#) ()
- virtual float [calculateDerivedParameter](#) (int index, vector< float > pars, float dt=1.0)

10.5.1 Constructor & Destructor Documentation

10.5.1.1 [dpclass::dpclass](#) () [inline]

10.5.2 Member Function Documentation

10.5.2.1 virtual float [dpclass::calculateDerivedParameter](#) (int *index*, vector< float > *pars*, float *dt* = 1.0) [inline], [virtual]

Reimplemented in [expDecayDp](#), [pwSTDP](#), [pwSTDP_userdef](#), [rulkovdp](#), and [pwSTDP](#).

The documentation for this class was generated from the following file:

- lib/include/[modelSpec.h](#)

10.6 errTupel Struct Reference

Public Attributes

- unsigned int [id](#)
- double [err](#)

10.6.1 Member Data Documentation

10.6.1.1 double [errTupel::err](#)

10.6.1.2 unsigned int [errTupel::id](#)

The documentation for this struct was generated from the following file:

- userproject/HHVclampGA_project/model/[GA.cc](#)

10.7 expDecayDp Class Reference

Class defining the dependent parameter for exponential decay.

```
#include <utils.h>
```

Inheritance diagram for [expDecayDp](#):

Public Member Functions

- float [calculateDerivedParameter](#) (int index, vector< float > pars, float dt=1.0)
- float [expDecay](#) (vector< float > pars, float dt)

10.7.1 Detailed Description

Class defining the dependent parameter for exponential decay.

10.7.2 Member Function Documentation

10.7.2.1 `float expDecayDp::calculateDerivedParameter (int index, vector< float > pars, float dt = 1.0) [inline], [virtual]`

Reimplemented from [dpclass](#).

10.7.2.2 `float expDecayDp::expDecay (vector< float > pars, float dt) [inline]`

The documentation for this class was generated from the following file:

- [lib/include/utils.h](#)

10.8 histogram Struct Reference

Public Attributes

- unsigned int [id](#)
- unsigned int [N](#)

10.8.1 Member Data Documentation

10.8.1.1 unsigned int `histogram::id`

10.8.1.2 unsigned int `histogram::N`

The documentation for this struct was generated from the following file:

- [userproject/MBody1_project/model/classol_sim_detailed_output.cu](#)

10.9 inputSpec Struct Reference

```
#include <helper.h>
```

Public Attributes

- double [t](#)
- double [baseV](#)
- int [N](#)
- vector< double > [st](#)
- vector< double > [V](#)

10.9.1 Member Data Documentation

10.9.1.1 double `inputSpec::baseV`

10.9.1.2 int `inputSpec::N`

10.9.1.3 vector<double> `inputSpec::st`

10.9.1.4 `double inputSpec::t`

10.9.1.5 `vector<double> inputSpec::V`

The documentation for this struct was generated from the following file:

- `userproject/HHVclampGA_project/model/helper.h`

10.10 neuronModel Struct Reference

class (struct) for specifying a neuron model.

```
#include <modelSpec.h>
```

Public Attributes

- string `simCode`
Code that defines the execution of one timestep of integration of the neuron model.
- string `thresholdConditionCode`
Code evaluating to a bool (e.g. "V > 20") that defines the condition for a true spike in the described neuron model.
- string `resetCode`
Code that defines the reset action taken after a spike occurred. This can be empty.
- vector< string > `varNames`
Names of the variables in the neuron model.
- vector< string > `tmpVarNames`
never used
- vector< string > `varTypes`
Types of the variable named above, e.g. "float". Names and types are matched by their order of occurrence in the vector.
- vector< string > `tmpVarTypes`
never used
- vector< string > `pNames`
Names of (independent) parameters of the model. These are assumed to be always of type "float".
- vector< string > `dpNames`
Names of dependent parameters of the model. These are assumed to be always of type "float".
- vector< string > `extraGlobalNeuronKernelParameters`
Additional parameter in the neuron kernel; it is translated to a population specific name but otherwise assumed to be one parameter per population rather than per neuron.
- vector< string > `extraGlobalNeuronKernelParameterTypes`
Additional parameters in the neuron kernel; they are translated to a population specific name but otherwise assumed to be one parameter per population rather than per neuron.
- `dpclass * dps`
- bool `needPreSt`
- bool `needPostSt`

10.10.1 Detailed Description

class (struct) for specifying a neuron model.

10.10.2 Member Data Documentation

10.10.2.1 `vector<string> neuronModel::dpNames`

Names of dependent parameters of the model. These are assumed to be always of type "float".

The dependent parameters are functions of independent parameters that enter into the neuron model. To avoid unnecessary computational overhead, these parameters are calculated at compile time and inserted as explicit values into the generated code. See method `NNmodel::initDerivedNeuronPara` for how this is done.

10.10.2.2 `dpclass* neuronModel::dps`

10.10.2.3 `vector<string> neuronModel::extraGlobalNeuronKernelParameters`

Additional parameter in the neuron kernel; it is translated to a population specific name but otherwise assumed to be one parameter per population rather than per neuron.

10.10.2.4 `vector<string> neuronModel::extraGlobalNeuronKernelParameterTypes`

Additional parameters in the neuron kernel; they are translated to a population specific name but otherwise assumed to be one parameter per population rather than per neuron.

10.10.2.5 `bool neuronModel::needPostSt`

10.10.2.6 `bool neuronModel::needPreSt`

10.10.2.7 `vector<string> neuronModel::pNames`

Names of (independent) parameters of the model. These are assumed to be always of type "float".

10.10.2.8 `string neuronModel::resetCode`

Code that defines the reset action taken after a spike occurred. This can be empty.

10.10.2.9 `string neuronModel::simCode`

Code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name "NN". It needs to refer to the predefined variable "ISYN", i.e. contain , if it is to receive input.

10.10.2.10 `string neuronModel::thresholdConditionCode`

Code evaluating to a bool (e.g. " $V > 20$ ") that defines the condition for a true spike in the described neuron model.

10.10.2.11 `vector<string> neuronModel::tmpVarNames`

never used

10.10.2.12 `vector<string> neuronModel::tmpVarTypes`

never used

10.10.2.13 `vector<string> neuronModel::varNames`

Names of the variables in the neuron model.

10.10.2.14 `vector<string> neuronModel::varTypes`

Types of the variable named above, e.g. "float". Names and types are matched by their order of occurrence in the vector.

The documentation for this struct was generated from the following file:

- `lib/include/modelSpec.h`

10.11 NNmodel Class Reference

Structure to hold the information that defines synapse dynamics (a model of how synapse variables change over time, independent of or in addition to changes when spikes occur).

```
#include <modelSpec.h>
```

Public Member Functions

- `NNmodel ()`
- `~NNmodel ()`
- void `setName` (const string)
Method to set the neuronal network model name.
- void `setPrecision` (unsigned int)
Set numerical precision for floating point.
- void `setTiming` (bool)
Set whether timers and timing commands are to be included.
- void `setSeed` (unsigned int)
Set the random seed (disables automatic seeding if argument not 0).
- void `checkSizes` (unsigned int *, unsigned int *, unsigned int *)
- void `resetPaddedSums` ()
Re-calculates the block-size-padded sum of threads needed to compute the groups of neurons and synapses assigned to each device. Must be called after changing the hostID:deviceId of any group.
- void `setGPUDevice` (int)
Method to choose the GPU to be used for the model. If "AUTODEVICE" (-1), GeNN will choose the device based on a heuristic rule.
- void `addNeuronPopulation` (const char *, unsigned int, unsigned int, float *, float *)
Method for adding a neuron population to a neuronal network model, using C style character array for the name of the population.
- void `addNeuronPopulation` (const string, unsigned int, unsigned int, float *, float *)
Method for adding a neuron population to a neuronal network model, using C++ string for the name of the population.
- void `activateDirectInput` (const string, unsigned int)
This function defines the type of the explicit input to the neuron model. Current options are common constant input to all neurons, input from a file and input defines as a rule.
- void `setConstInp` (const string, float)
Method for setting the global input value for a neuron population if CONSTINP.
- void `setNeuronClusterIndex` (const string neuronGroup, int hostID, int deviceId)
Function for setting which host and which device a neuron group will be simulated on.
- void `addSynapsePopulation` (const string name, unsigned int syntype, unsigned int conntype, unsigned int gtype, const string src, const string trg, float *p)

Overload of method for backwards compatibility.

- void [addSynapsePopulation](#) (const char *, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, const char *, const char *, float *, float *, float *)

Method for adding a synapse population to a neuronal network model, using C style character array for the name of the population.

- void [addSynapsePopulation](#) (const string, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, const string, const string, float *, float *, float *)

Overloaded version without initial variables for synapses.

- void [addSynapsePopulation](#) (const string, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, const string, const string, float *, float *, float *, float *)

Method for adding a synapse population to a neuronal network model, using C++ string for the name of the population.

- void [setSynapseG](#) (const string, float)

Method for setting the conductance (g) value for a synapse population with "GLOBALG" characteristic.

- void [setMaxConn](#) (const string, unsigned int)

This function defines the maximum number of connections for a neuron in the population.

- void [setSynapseClusterIndex](#) (const string synapseGroup, int hostID, int deviceID)

Function for setting which host and which device a synapse group will be simulated on.

- void [initLearnGrps](#) ()

Public Attributes

- string [name](#)

Name of the neuronal newtwork model.

- string [ftype](#)

Type of floating point variables (float, double, ...; default: float)

- string [RNtype](#)

Underlying type for random number generation (default: long)

- int [valid](#)

Flag for whether the model has been validated (unused?)

- unsigned int [needSt](#)

Whether last spike times are needed at all in this network model (related to STDP)

- unsigned int [needSynapseDelay](#)

Whether delayed synapse conductance is required in the network.

- int [chooseGPUDevice](#)

- bool [timing](#)

- unsigned int [seed](#)

- bool [needSpkEvt](#)

- vector< string > [neuronName](#)

Names of neuron groups.

- unsigned int [neuronGrpN](#)

Number of neuron groups.

- vector< unsigned int > [neuronN](#)

Number of neurons in group.

- vector< unsigned int > [sumNeuronN](#)

Summed neuron numbers.

- vector< unsigned int > [padSumNeuronN](#)

Padded summed neuron numbers.

- vector< unsigned int > [neuronPostSyn](#)

- vector< unsigned int > [neuronType](#)

Postsynaptic methods to the neuron.

- vector< vector< float > > [neuronPara](#)

Parameters of neurons.

- vector< vector< float > > [dnp](#)

Derived neuron parameters.

- vector< vector< float > > [neuronIni](#)

Initial values of neurons.

- vector< vector< unsigned int > > [inSyn](#)

The ids of the incoming synapse groups.

- vector< vector< unsigned int > > [outSyn](#)

The ids of the outgoing synapse groups.

- vector< unsigned int > [receivesInputCurrent](#)

flags whether neurons of a population receive explicit input currents

- vector< bool > [neuronNeedSt](#)

Whether last spike time needs to be saved for each individual neuron type.

- vector< bool > [neuronNeedSpkEvt](#)

Whether this neuron group needs to record spike like events.

- vector< string > [neuronSpkEvtCondition](#)

Will contain the spike event condition code when spike events are used.

- vector< unsigned int > [neuronDelaySlots](#)

The number of slots needed in the synapse delay queues of a neuron group.

- vector< int > [neuronHostID](#)

The ID of the cluster node which the neuron groups are computed on.

- vector< int > [neuronDeviceID](#)

The ID of the CUDA device which the neuron groups are computed on.

- vector< vector< bool > > [neuronVarNeedSpkEvt](#)

indicates whether spkEnt values (or delay queues) need to be stored for this variable

- vector< vector< bool > > [neuronVarNeedSpk](#)

indicates whether spk values (or delay queues) need to be stored for this variable

- vector< string > [synapseName](#)

Names of synapse groups.

- unsigned int [synapseGrpN](#)

Number of synapse groups.

- vector< unsigned int > [sumSynapseTrgN](#)

Summed number of target neurons.

- vector< unsigned int > [padSumSynapseTrgN](#)

"Padded" summed target neuron numbers

- vector< unsigned int > [maxConn](#)

Padded summed maximum number of connections for a neuron in the neuron groups.

- vector< unsigned int > [padSumSynapseKrnI](#)

- vector< unsigned int > [synapseType](#)

Types of synapses.

- vector< unsigned int > [synapseConnType](#)

Connectivity type of synapses.

- vector< unsigned int > [synapseGType](#)

Type of specification method for synaptic conductance.

- vector< unsigned int > [synapseSource](#)

Presynaptic neuron groups.

- vector< unsigned int > [synapseTarget](#)

Postsynaptic neuron groups.

- vector< unsigned int > [synapseInSynNo](#)

IDs of the target neurons' incoming synapse variables for each synapse group.

- vector< unsigned int > [synapseOutSynNo](#)

The target neurons' outgoing synapse for each synapse group.

- vector< unsigned int > [usesTrueSpikes](#)

Defines if synapse update is done after detection of real spikes (only one point after threshold)

- vector< unsigned int > [usesSpikeEvents](#)

Defines if synapse update is done after detection of spike events (every point above threshold)

- vector< vector< string > > [synapseSpkEvtVars](#)

Defines variable names that are needed in the SpkEvt condition and that are pre-fetched for that purpose into shared memory.

- vector< unsigned int > [usesPostLearning](#)

Defines if anything is done in case of postsynaptic neuron spiking before presynaptic neuron (punishment in STDP etc.)

- vector< vector< float > > [synapsePara](#)

parameters of synapses

- vector< vector< float > > [synapseIni](#)

Initial values of synapse variables.

- vector< vector< float > > [dsp_w](#)

Derived synapse parameters ([weightUpdateModel](#) only)

- vector< unsigned int > [postSynapseType](#)

Types of post-synaptic model.

- vector< vector< float > > [postSynapsePara](#)

parameters of postsynapses

- vector< vector< float > > [postSynIni](#)

Initial values of postsynaptic variables.

- vector< vector< float > > [dpsp](#)

Derived postsynapse parameters.

- vector< float > [globalInp](#)

Global explicit input if CONSTINP is chosen.

- unsigned int [lrnGroups](#)

Number of synapse groups with learning.

- vector< unsigned int > [padSumLearnN](#)

Padded summed neuron numbers of learn group source populations.

- vector< unsigned int > [lrnSynGrp](#)

Enumeration of the IDs of synapse groups that learn.

- vector< unsigned int > [synapseDelay](#)

Global synaptic conductance delay for the group (in time steps)

- vector< int > [synapseHostID](#)

The ID of the cluster node which the synapse groups are computed on.

- vector< int > [synapseDeviceID](#)

The ID of the CUDA device which the synapse groups are computed on.

10.11.1 Detailed Description

Structure to hold the information that defines synapse dynamics (a model of how synapse variables change over time, independent of or in addition to changes when spikes occur).

10.11.2 Constructor & Destructor Documentation

10.11.2.1 `NNmodel::NNmodel ()`

10.11.2.2 `NNmodel::~~NNmodel ()`

10.11.3 Member Function Documentation

10.11.3.1 `void NNmodel::activateDirectInput (const string name, unsigned int type)`

This function defines the type of the explicit input to the neuron model. Current options are common constant input to all neurons, input from a file and input defines as a rule.

Parameters

<i>name</i>	Name of the neuron population
<i>type</i>	Type of input: 1 if common input, 2 if custom input from file, 3 if custom input as a rule

10.11.3.2 `void NNmodel::addNeuronPopulation (const char * name, unsigned int nNo, unsigned int type, float * p, float * ini)`

Method for adding a neuron population to a neuronal network model, using C style character array for the name of the population.

Parameters

<i>name</i>	Name of the neuron population
<i>nNo</i>	Number of neurons in the population
<i>type</i>	Type of the neurons, refers to either a standard type or user-defined type
<i>p</i>	Parameters of this neuron type
<i>ini</i>	Initial values for variables of this neuron type

10.11.3.3 `void NNmodel::addNeuronPopulation (const string name, unsigned int nNo, unsigned int type, float * p, float * ini)`

Method for adding a neuron population to a neuronal network model, using C++ string for the name of the population.

This function adds a neuron population to a neuronal network models, assigning the name, the number of neurons in the group, the neuron type, parameters and initial values.

Parameters

<i>name</i>	The name of the neuron population
<i>nNo</i>	Number of neurons in the population
<i>type</i>	Type of the neurons, refers to either a standard type or user-defined type
<i>p</i>	Parameters of this neuron type
<i>ini</i>	Initial values for variables of this neuron type

10.11.3.4 `void NNmodel::addSynapsePopulation (const string name, unsigned int syntype, unsigned int conntype, unsigned int gtype, const string src, const string target, float * params)`

Overload of method for backwards compatibility.

Parameters

<i>name</i>	The name of the synapse population
<i>syntype</i>	The type of synapse to be added (i.e. learning mode)
<i>conntype</i>	The type of synaptic connectivity
<i>gtype</i>	The way how the synaptic conductivity g will be defined
<i>src</i>	Name of the (existing!) pre-synaptic neuron population
<i>target</i>	Name of the (existing!) post-synaptic neuron population
<i>params</i>	A C-type array of floats that contains synapse parameter values (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.

10.11.3.5 void NNmodel::addSynapsePopulation (const char * *name*, unsigned int *syntype*, unsigned int *conntype*, unsigned int *gtype*, unsigned int *delaySteps*, unsigned int *postsyn*, const char * *src*, const char * *trg*, float * *p*, float * *PSVini*, float * *ps*)

Method for adding a synapse population to a neuronal network model, using C style character array for the name of the population.

Parameters

<i>name</i>	The name of the synapse population
<i>syntype</i>	The type of synapse to be added (i.e. learning mode)
<i>conntype</i>	The type of synaptic connectivity
<i>gtype</i>	The way how the synaptic conductivity g will be defined
<i>delaySteps</i>	Number of delay slots
<i>postsyn</i>	Postsynaptic integration method
<i>src</i>	Name of the (existing!) pre-synaptic neuron population
<i>trg</i>	Name of the (existing!) post-synaptic neuron population
<i>p</i>	A C-type array of floats that contains synapse parameter values (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.
<i>PSVini</i>	A C-type array of floats that contains the initial values for postsynaptic mechanism variables (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.
<i>ps</i>	A C-type array of floats that contains postsynaptic mechanism parameter values (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.

10.11.3.6 void NNmodel::addSynapsePopulation (const string *name*, unsigned int *syntype*, unsigned int *conntype*, unsigned int *gtype*, unsigned int *delaySteps*, unsigned int *postsyn*, const string *src*, const string *trg*, float * *p*, float * *PSVini*, float * *ps*)

Overloaded version without initial variables for synapses.

Overloaded old version.

Parameters

<i>name</i>	The name of the synapse population
<i>syntype</i>	The type of synapse to be added (i.e. learning mode)
<i>conntype</i>	The type of synaptic connectivity
<i>gtype</i>	The way how the synaptic conductivity <i>g</i> will be defined
<i>delaySteps</i>	Number of delay slots
<i>postsyn</i>	Postsynaptic integration method
<i>src</i>	Name of the (existing!) pre-synaptic neuron population
<i>trg</i>	Name of the (existing!) post-synaptic neuron population
<i>p</i>	A C-type array of floats that contains synapse parameter values (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.
<i>PSVini</i>	A C-type array of floats that contains the initial values for postsynaptic mechanism variables (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.
<i>ps</i>	A C-type array of floats that contains postsynaptic mechanism parameter values (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.

10.11.3.7 void NNmodel::addSynapsePopulation (const string *name*, unsigned int *syntype*, unsigned int *conntype*, unsigned int *gtype*, unsigned int *delaySteps*, unsigned int *postsyn*, const string *src*, const string *trg*, float * *synini*, float * *p*, float * *PSVini*, float * *ps*)

Method for adding a synapse population to a neuronal network model, using C++ string for the name of the population.

This function adds a synapse population to a neuronal network model, assigning the name, the synapse type, the connectivity type, the type of conductance specification, the source and destination neuron populations, and the synaptic parameters.

Parameters

<i>name</i>	The name of the synapse population
<i>syntype</i>	The type of synapse to be added (i.e. learning mode)
<i>conntype</i>	The type of synaptic connectivity
<i>gtype</i>	The way how the synaptic conductivity <i>g</i> will be defined
<i>delaySteps</i>	Number of delay slots
<i>postsyn</i>	Postsynaptic integration method
<i>src</i>	Name of the (existing!) pre-synaptic neuron population
<i>trg</i>	Name of the (existing!) post-synaptic neuron population
<i>synini</i>	A C-type array of floats that contains the initial values for synapse variables (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.
<i>p</i>	A C-type array of floats that contains synapse parameter values (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.

<i>PSVini</i>	A C-type array of floats that contains the initial values for postsynaptic mechanism variables (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.
<i>ps</i>	A C-type array of floats that contains postsynaptic mechanism parameter values (common to all synapses of the population) which will be used for the defined synapses. The array must contain the right number of parameters in the right order for the chosen synapse type. If too few, segmentation faults will occur, if too many, excess will be ignored.

10.11.3.8 void NNmodel::checkSizes (unsigned int *, unsigned int *, unsigned int *)

10.11.3.9 void NNmodel::initLearnGrps ()

10.11.3.10 void NNmodel::resetPaddedSums ()

Re-calculates the block-size-padded sum of threads needed to compute the groups of neurons and synapses assigned to each device. Must be called after changing the hostID:deviceID of any group.

This function re-calculates the block-size-padded sum of threads needed to compute the groups of neurons and synapses assigned to each device. Must be called after changing the hostID:deviceID of any neuron or synapse group.

10.11.3.11 void NNmodel::setConstInp (const string sName, float globalInp0)

Method for setting the global input value for a neuron population if CONSTINP.

This function sets a global input value to the specified neuron group.

10.11.3.12 void NNmodel::setGPUDevice (int device)

Method to choose the GPU to be used for the model. If "AUTODEVICE" (-1), GeNN will choose the device based on a heuristic rule.

This function defines the way how the GPU is chosen. If "AUTODEVICE" (-1) is given as the argument, GeNN will use internal heuristics to choose the device. Otherwise the argument is the device number and the indicated device will be used.

10.11.3.13 void NNmodel::setMaxConn (const string sname, unsigned int maxConnP)

This function defines the maximum number of connections for a neuron in the population.

10.11.3.14 void NNmodel::setName (const string inname)

Method to set the neuronal network model name.

10.11.3.15 void NNmodel::setNeuronClusterIndex (const string neuronGroup, int hostID, int deviceID)

Function for setting which host and which device a neuron group will be simulated on.

This function is for setting which host and which device a neuron group will be simulated on.

Parameters

<i>neuronGroup</i>	Name of the neuron population
<i>hostID</i>	ID of the host
<i>deviceID</i>	ID of the device

10.11.3.16 void NNmodel::setPrecision (unsigned int *floattype*)

Set numerical precision for floating point.

This function sets the numerical precision of floating type variables. By default, it is float.

10.11.3.17 void NNmodel::setSeed (unsigned int *inseed*)

Set the random seed (disables automatic seeding if argument not 0).

This function sets the random seed. If the passed argument is > 0, automatic seeding is disabled. If the argument is 0, the underlying seed is obtained from the time() function.

Parameters

<i>inseed</i>	the new seed
---------------	--------------

10.11.3.18 void NNmodel::setSynapseClusterIndex (const string *synapseGroup*, int *hostID*, int *deviceID*)

Function for setting which host and which device a synapse group will be simulated on.

This function is for setting which host and which device a synapse group will be simulated on.

Parameters

<i>synapseGroup</i>	Name of the synapse population
<i>hostID</i>	ID of the host
<i>deviceID</i>	ID of the device

10.11.3.19 void NNmodel::setSynapseG (const string *sName*, float *g*)

Method for setting the conductance (g) value for a synapse population with "GLOBALG" characteristic.

This functions sets the global value of the maximal synaptic conductance for a synapse population that was identified as conductance specification method "GLOBALG".

10.11.3.20 void NNmodel::setTiming (bool *theTiming*)

Set whether timers and timing commands are to be included.

This function sets a flag to determine whether timers and timing commands are to be included in generated code.

10.11.4 Member Data Documentation

10.11.4.1 int NNmodel::chooseGPUDevice

10.11.4.2 vector<vector<float>> NNmodel::dnp

Derived neuron parameters.

10.11.4.3 `vector<vector<float> > NNmodel::dpsp`

Derived postsynapse parameters.

10.11.4.4 `vector<vector<float> > NNmodel::dsp_w`

Derived synapse parameters ([weightUpdateModel](#) only)

10.11.4.5 `string NNmodel::ftype`

Type of floating point variables (float, double, ...; default: float)

10.11.4.6 `vector<float> NNmodel::globalInp`

Global explicit input if CONSTINP is chosen.

10.11.4.7 `vector<vector<unsigned int> > NNmodel::inSyn`

The ids of the incoming synapse groups.

10.11.4.8 `unsigned int NNmodel::lrnGroups`

Number of synapse groups with learning.

10.11.4.9 `vector<unsigned int> NNmodel::lrnSynGrp`

Enumeration of the IDs of synapse groups that learn.

10.11.4.10 `vector<unsigned int> NNmodel::maxConn`

Padded summed maximum number of connections for a neuron in the neuron groups.

10.11.4.11 `string NNmodel::name`

Name of the neuronal newtwork model.

10.11.4.12 `bool NNmodel::needSpkEvt`

10.11.4.13 `unsigned int NNmodel::needSt`

Whether last spike times are needed at all in this network model (related to STDP)

10.11.4.14 `unsigned int NNmodel::needSynapseDelay`

Whether delayed synapse conductance is required in the network.

10.11.4.15 `vector<unsigned int> NNmodel::neuronDelaySlots`

The number of slots needed in the synapse delay queues of a neuron group.

10.11.4.16 `vector<int> NNmodel::neuronDeviceID`

The ID of the CUDA device which the neuron groups are computed on.

10.11.4.17 `unsigned int NNmodel::neuronGrpN`

Number of neuron groups.

10.11.4.18 `vector<int> NNmodel::neuronHostID`

The ID of the cluster node which the neuron groups are computed on.

10.11.4.19 `vector<vector<float> > NNmodel::neuronIni`

Initial values of neurons.

10.11.4.20 `vector<unsigned int> NNmodel::neuronN`

Number of neurons in group.

10.11.4.21 `vector<string> NNmodel::neuronName`

Names of neuron groups.

10.11.4.22 `vector<bool> NNmodel::neuronNeedSpkEvt`

Whether this neuron group needs to record spike like events.

10.11.4.23 `vector<bool> NNmodel::neuronNeedSt`

Whether last spike time needs to be saved for each individual neuron type.

10.11.4.24 `vector<vector<float> > NNmodel::neuronPara`

Parameters of neurons.

10.11.4.25 `vector<unsigned int> NNmodel::neuronPostSyn`

10.11.4.26 `vector<string> NNmodel::neuronSpkEvtCondition`

Will contain the spike event condition code when spike events are used.

10.11.4.27 `vector<unsigned int> NNmodel::neuronType`

Postsynaptic methods to the neuron.

Types of neurons

10.11.4.28 `vector<vector<bool>>> NNmodel::neuronVarNeedSpk`

indicates whether spk values (or delay queues) need to be stored for this variable

10.11.4.29 `vector<vector<bool>>> NNmodel::neuronVarNeedSpkEvt`

indicates whether spkEnt values (or delay queues) need to be stored for this variable

10.11.4.30 `vector<vector<unsigned int>>> NNmodel::outSyn`

The ids of the outgoing synapse groups.

10.11.4.31 `vector<unsigned int> NNmodel::padSumLearnN`

Padded summed neuron numbers of learn group source populations.

10.11.4.32 `vector<unsigned int> NNmodel::padSumNeuronN`

Padded summed neuron numbers.

10.11.4.33 `vector<unsigned int> NNmodel::padSumSynapseKrnl`

10.11.4.34 `vector<unsigned int> NNmodel::padSumSynapseTrgN`

"Padded" summed target neuron numbers

10.11.4.35 `vector<vector<float>>> NNmodel::postSynapsePara`

parameters of postsynapses

10.11.4.36 `vector<unsigned int> NNmodel::postSynapseType`

Types of post-synaptic model.

10.11.4.37 `vector<vector<float>>> NNmodel::postSynIni`

Initial values of postsynaptic variables.

10.11.4.38 `vector<unsigned int> NNmodel::receivesInputCurrent`

flags whether neurons of a population receive explicit input currents

10.11.4.39 `string NNmodel::RNtype`

Underlying type for random number generation (default: long)

10.11.4.40 `unsigned int NNmodel::seed`

10.11.4.41 `vector<unsigned int> NNmodel::sumNeuronN`

Summed neuron numbers.

10.11.4.42 `vector<unsigned int> NNmodel::sumSynapseTrgN`

Summed number of target neurons.

10.11.4.43 `vector<unsigned int> NNmodel::synapseConnType`

Connectivity type of synapses.

10.11.4.44 `vector<unsigned int> NNmodel::synapseDelay`

Global synaptic conductance delay for the group (in time steps)

10.11.4.45 `vector<int> NNmodel::synapseDeviceID`

The ID of the CUDA device which the synapse groups are computed on.

10.11.4.46 `unsigned int NNmodel::synapseGrpN`

Number of synapse groups.

10.11.4.47 `vector<unsigned int> NNmodel::synapseGType`

Type of specification method for synaptic conductance.

10.11.4.48 `vector<int> NNmodel::synapseHostID`

The ID of the cluster node which the synapse groups are computed on.

10.11.4.49 `vector<vector<float>> NNmodel::synapseIni`

Initial values of synapse variables.

10.11.4.50 `vector<unsigned int> NNmodel::synapseInSynNo`

IDs of the target neurons' incoming synapse variables for each synapse group.

10.11.4.51 `vector<string> NNmodel::synapseName`

Names of synapse groups.

10.11.4.52 `vector<unsigned int> NNmodel::synapseOutSynNo`

The target neurons' outgoing synapse for each synapse group.

10.11.4.53 `vector<vector<float>> NNmodel::synapsePara`

parameters of synapses

10.11.4.54 `vector<unsigned int> NNmodel::synapseSource`

Presynaptic neuron groups.

10.11.4.55 `vector<vector<string>> NNmodel::synapseSpkEvtVars`

Defines variable names that are needed in the SpkEvt condition and that are pre-fetched for that purpose into shared memory.

10.11.4.56 `vector<unsigned int> NNmodel::synapseTarget`

Postsynaptic neuron groups.

10.11.4.57 `vector<unsigned int> NNmodel::synapseType`

Types of synapses.

10.11.4.58 `bool NNmodel::timing`

10.11.4.59 `vector<unsigned int> NNmodel::usesPostLearning`

Defines if anything is done in case of postsynaptic neuron spiking before presynaptic neuron (punishment in STDP etc.)

10.11.4.60 `vector<unsigned int> NNmodel::usesSpikeEvents`

Defines if synapse update is done after detection of spike events (every point above threshold)

10.11.4.61 `vector<unsigned int> NNmodel::usesTrueSpikes`

Defines if synapse update is done after detection of real spikes (only one point after threshold)

10.11.4.62 `int NNmodel::valid`

Flag for whether the model has been validated (unused?)

The documentation for this class was generated from the following files:

- [lib/include/modelSpec.h](#)
- [lib/include/modelSpec.cc](#)

10.12 postSynModel Struct Reference

Structure to hold the information that defines a post-synaptic model (a model of how synapses affect post-synaptic neuron variables, classically in the form of a synaptic current). It also allows to define an equation for the dynamics that can be applied to the summed synaptic input variable "insyn".

```
#include <modelSpec.h>
```


Public Attributes

- string [postSyntoCurrent](#)
- string [postSynDecay](#)
- vector< string > [varNames](#)
Names of the variables in the postsynaptic model.
- vector< string > [varTypes](#)
Types of the variable named above, e.g. "float". Names and types are matched by their order of occurrence in the vector.
- vector< string > [pNames](#)
Names of (independent) parameters of the model. These are assumed to be always of type "float".
- vector< string > [dpNames](#)
Names of dependent parameters of the model. These are assumed to be always of type "float".
- [dpclass](#) * [dps](#)

10.12.1 Detailed Description

Structure to hold the information that defines a post-synaptic model (a model of how synapses affect post-synaptic neuron variables, classically in the form of a synaptic current). It also allows to define an equation for the dynamics that can be applied to the summed synaptic input variable "insyn".

10.12.2 Member Data Documentation

10.12.2.1 vector<string> postSynModel::dpNames

Names of dependent parameters of the model. These are assumed to be always of type "float".

10.12.2.2 dpclass* postSynModel::dps

10.12.2.3 vector<string> postSynModel::pNames

Names of (independent) parameters of the model. These are assumed to be always of type "float".

10.12.2.4 string postSynModel::postSynDecay

10.12.2.5 string postSynModel::postSyntoCurrent

10.12.2.6 vector<string> postSynModel::varNames

Names of the variables in the postsynaptic model.

10.12.2.7 vector<string> postSynModel::varTypes

Types of the variable named above, e.g. "float". Names and types are matched by their order of occurrence in the vector.

The documentation for this struct was generated from the following file:

- lib/include/[modelSpec.h](#)

10.13 pwSTDP Class Reference

TODO This class definition may be code-generated in a future release.

```
#include <utils.h>
```

Inheritance diagram for pwSTDP:

Public Member Functions

- float [calculateDerivedParameter](#) (int index, vector< float > pars, float dt)
- float [lim0](#) (vector< float > pars, float dt)
- float [lim1](#) (vector< float > pars, float dt)
- float [slope0](#) (vector< float > pars, float dt)
- float [slope1](#) (vector< float > pars, float dt)
- float [off0](#) (vector< float > pars, float dt)
- float [off1](#) (vector< float > pars, float dt)
- float [off2](#) (vector< float > pars, float dt)
- float [calculateDerivedParameter](#) (int index, vector< float > pars, float dt=[DT](#))
- float [lim0](#) (vector< float > pars, float dt)
- float [lim1](#) (vector< float > pars, float dt)
- float [slope0](#) (vector< float > pars, float dt)
- float [slope1](#) (vector< float > pars, float dt)
- float [off0](#) (vector< float > pars, float dt)
- float [off1](#) (vector< float > pars, float dt)
- float [off2](#) (vector< float > pars, float dt)

10.13.1 Detailed Description

TODO This class definition may be code-generated in a future release.

This class defines derived parameters for the learn1synapse standard weightupdate model

10.13.2 Member Function Documentation

10.13.2.1 float pwSTDP::calculateDerivedParameter (int *index*, vector< float > *pars*, float *dt*) [inline],
[virtual]

Reimplemented from [dpclass](#).

10.13.2.2 float pwSTDP::calculateDerivedParameter (int *index*, vector< float > *pars*, float *dt* = [DT](#)) [inline],
[virtual]

Reimplemented from [dpclass](#).

10.13.2.3 float pwSTDP::lim0 (vector< float > *pars*, float *dt*) [inline]

10.13.2.4 float pwSTDP::lim0 (vector< float > *pars*, float *dt*) [inline]

10.13.2.5 float pwSTDP::lim1 (vector< float > *pars*, float *dt*) [inline]

10.13.2.6 float pwSTDP::lim1 (vector< float > *pars*, float *dt*) [inline]

- 10.13.2.7 float pwSTDP::off0 (vector< float > *pars*, float *dt*) [inline]
- 10.13.2.8 float pwSTDP::off0 (vector< float > *pars*, float *dt*) [inline]
- 10.13.2.9 float pwSTDP::off1 (vector< float > *pars*, float *dt*) [inline]
- 10.13.2.10 float pwSTDP::off1 (vector< float > *pars*, float *dt*) [inline]
- 10.13.2.11 float pwSTDP::off2 (vector< float > *pars*, float *dt*) [inline]
- 10.13.2.12 float pwSTDP::off2 (vector< float > *pars*, float *dt*) [inline]
- 10.13.2.13 float pwSTDP::slope0 (vector< float > *pars*, float *dt*) [inline]
- 10.13.2.14 float pwSTDP::slope0 (vector< float > *pars*, float *dt*) [inline]
- 10.13.2.15 float pwSTDP::slope1 (vector< float > *pars*, float *dt*) [inline]
- 10.13.2.16 float pwSTDP::slope1 (vector< float > *pars*, float *dt*) [inline]

The documentation for this class was generated from the following files:

- lib/include/[utils.h](#)
- tmp/model/[MBody_userdef.cc](#)

10.14 pwSTDP_userdef Class Reference

TODO This class definition may be code-generated in a future release.

Inheritance diagram for pwSTDP_userdef:

Public Member Functions

- float [calculateDerivedParameter](#) (int *index*, vector< float > *pars*, float *dt*=[DT](#))
- float [lim0](#) (vector< float > *pars*, float *dt*)
- float [lim1](#) (vector< float > *pars*, float *dt*)
- float [slope0](#) (vector< float > *pars*, float *dt*)
- float [slope1](#) (vector< float > *pars*, float *dt*)
- float [off0](#) (vector< float > *pars*, float *dt*)
- float [off1](#) (vector< float > *pars*, float *dt*)
- float [off2](#) (vector< float > *pars*, float *dt*)

10.14.1 Detailed Description

TODO This class definition may be code-generated in a future release.

10.14.2 Member Function Documentation

- 10.14.2.1 float pwSTDP_userdef::calculateDerivedParameter (int *index*, vector< float > *pars*, float *dt* = [DT](#)) [inline],
[virtual]

Reimplemented from [dpclass](#).

10.14.2.2 float pwSTDP_userdef::lim0 (vector< float > *pars*, float *dt*) [inline]

10.14.2.3 float pwSTDP_userdef::lim1 (vector< float > *pars*, float *dt*) [inline]

10.14.2.4 float pwSTDP_userdef::off0 (vector< float > *pars*, float *dt*) [inline]

10.14.2.5 float pwSTDP_userdef::off1 (vector< float > *pars*, float *dt*) [inline]

10.14.2.6 float pwSTDP_userdef::off2 (vector< float > *pars*, float *dt*) [inline]

10.14.2.7 float pwSTDP_userdef::slope0 (vector< float > *pars*, float *dt*) [inline]

10.14.2.8 float pwSTDP_userdef::slope1 (vector< float > *pars*, float *dt*) [inline]

The documentation for this class was generated from the following file:

- [userproject/MBody_userdef_project/model/MBody_userdef.cc](#)

10.15 randomGauss Class Reference

Class random Gauss encapsulates the methods for generating random neumbers with Gaussian distribution.

```
#include <gauss.h>
```

Public Member Functions

- [randomGauss](#) ()
Constructor for the Gaussian random number generator class without giving explicit seeds.
- [randomGauss](#) (unsigned long, unsigned long, unsigned long)
Constructor for the Gaussian random number generator class when seeds are provided explicitly.
- [~randomGauss](#) ()
- double [n](#) ()
Method for obtaining a random number with Gaussian distribution.

10.15.1 Detailed Description

Class random Gauss encapsulates the methods for generating random neumbers with Gaussian distribution.

A random number from a Gaussian distribution of mean 0 and standard deviation 1 is obtained by calling the method [randomGauss::n\(\)](#).

10.15.2 Constructor & Destructor Documentation

10.15.2.1 [randomGauss::randomGauss](#) () [explicit]

Constructor for the Gaussian random number generator class without giving explicit seeds.

The seeds for random number generation are generated from the internal clock of the computer during execution.

10.15.2.2 [randomGauss::randomGauss](#) (unsigned long *seed1*, unsigned long *seed2*, unsigned long *seed3*)

Constructor for the Gaussian random number generator class when seeds are provided explicitly.

The seeds are three arbitrary unsigned long integers.

10.15.2.3 `randomGauss::~~randomGauss () [inline]`

10.15.3 Member Function Documentation

10.15.3.1 `double randomGauss::n ()`

Method for obtaining a random number with Gaussian distribution.

Function for generating a pseudo random number from a Gaussian distribution.

The documentation for this class was generated from the following files:

- `lib/include/numlib/gauss.h`
- `lib/include/numlib/gauss.cc`

10.16 randomGen Class Reference

Class `randomGen` which implements the ISAAC random number generator for uniformly distributed random numbers.

```
#include <randomGen.h>
```

Public Member Functions

- `randomGen ()`
Constructor for the ISAAC random number generator class without giving explicit seeds.
- `randomGen (unsigned long, unsigned long, unsigned long)`
Constructor for the Gaussian random number generator class when seeds are provided explicitly.
- `~randomGen ()`
- `double n ()`
Method to obtain a random number from a uniform ditribution on [0,1].

10.16.1 Detailed Description

Class `randomGen` which implements the ISAAC random number generator for uniformly distributed random numbers.

The random number generator initializes with system timea or explicit seeds and returns a random number according to a uniform distribution on [0,1]; making use of the ISAAC random number generator; C++ Implementation by Quinn Tyler Jackson of the RG invented by Bob Jenkins Jr.

10.16.2 Constructor & Destructor Documentation

10.16.2.1 `randomGen::randomGen () [explicit]`

Constructor for the ISAAC random number generator class without giving explicit seeds.

The seeds for random number generation are generated from the internal clock of the computer during execution.

10.16.2.2 `randomGen::randomGen (unsigned long seed1, unsigned long seed2, unsigned long seed3)`

Constructor for the Gaussian random number generator class when seeds are provided explicitly.

The seeds are three arbitrary unsigned long integers.

10.16.2.3 `randomGen::~~randomGen () [inline]`

10.16.3 Member Function Documentation

10.16.3.1 `double randomGen::n ()`

Method to obtain a random number from a uniform ditribution on [0,1].

Function for generating a pseudo random number from a uniform distribution on the interval [0,1].

The documentation for this class was generated from the following files:

- lib/include/numlib/[randomGen.h](#)
- lib/include/numlib/[randomGen.cc](#)

10.17 rulkovdp Class Reference

Class defining the dependent parameters of teh Rulkov map neuron.

```
#include <utils.h>
```

Inheritance diagram for rulkovdp:

Public Member Functions

- float [calculateDerivedParameter](#) (int *index*, vector< float > *pars*, float *dt*=1.0)
- float [ip0](#) (vector< float > *pars*)
- float [ip1](#) (vector< float > *pars*)
- float [ip2](#) (vector< float > *pars*)

10.17.1 Detailed Description

Class defining the dependent parameters of teh Rulkov map neuron.

10.17.2 Member Function Documentation

10.17.2.1 `float rulkovdp::calculateDerivedParameter (int index, vector< float > pars, float dt = 1.0) [inline], [virtual]`

Reimplemented from [dpclass](#).

10.17.2.2 `float rulkovdp::ip0 (vector< float > pars) [inline]`

10.17.2.3 `float rulkovdp::ip1 (vector< float > pars) [inline]`

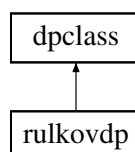
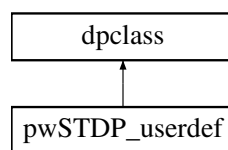
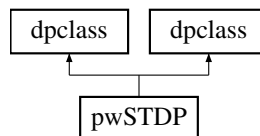
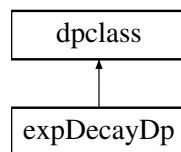
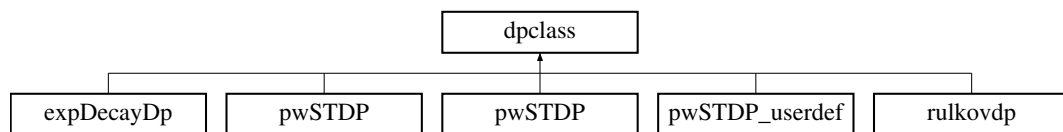
10.17.2.4 `float rulkovdp::ip2 (vector< float > pars) [inline]`

The documentation for this class was generated from the following file:

- lib/include/[utils.h](#)

10.18 stdRG Class Reference

```
#include <randomGen.h>
```



Public Member Functions

- [stdRG](#) ()
Constructor of the standard random number generator class without explicit seed.
- [stdRG](#) (unsigned int)
Constructor of the standard random number generator class with explicit seed.
- [~stdRG](#) ()
- double [n](#) ()
Method to generate a uniform random number.
- unsigned long [nlong](#) ()

10.18.1 Constructor & Destructor Documentation

10.18.1.1 `stdRG::stdRG () [explicit]`

Constructor of the standard random number generator class without explicit seed.

The seed is taken from teh internal clock of the computer.

10.18.1.2 `stdRG::stdRG (unsigned int seed)`

Constructor of the standard random number generator class with explicit seed.

The seed is an arbitrary unsigned int

10.18.1.3 `stdRG::~stdRG () [inline]`

10.18.2 Member Function Documentation

10.18.2.1 `double stdRG::n ()`

Method to generate a uniform random number.

The moethod is a wrapper for the C function rand() and returns a pseudo random number in the interval [0,1[

10.18.2.2 `unsigned long stdRG::nlong ()`

The documentation for this class was generated from the following files:

- lib/include/numlib/[randomGen.h](#)
- lib/include/numlib/[randomGen.cc](#)

10.19 stopWatch Struct Reference

```
#include <hr_time.h>
```

Public Attributes

- timeval [start](#)
- timeval [stop](#)

10.19.1 Member Data Documentation

10.19.1.1 `timeval stopWatch::start`

10.19.1.2 `timeval stopWatch::stop`

The documentation for this struct was generated from the following file:

- [lib/include/hr_time.h](#)

10.20 SynDelay Class Reference

```
#include <SynDelaySim.h>
```

Public Member Functions

- [SynDelay](#) (bool usingGPU)
- [~SynDelay](#) ()
- void [run](#) (float t)

10.20.1 Constructor & Destructor Documentation

10.20.1.1 `SynDelay::SynDelay (bool usingGPU)`

10.20.1.2 `SynDelay::~~SynDelay ()`

10.20.2 Member Function Documentation

10.20.2.1 `void SynDelay::run (float t)`

The documentation for this class was generated from the following files:

- [userproject/SynDelay_project/SynDelaySim.h](#)
- [userproject/SynDelay_project/SynDelaySim.cu](#)

10.21 weightUpdateModel Class Reference

Structure to hold the information that defines a weightupdate model (a model of how spikes affect synaptic (and/or) (mostly) post-synaptic neuron variables. It also allows to define changes in response to post-synaptic spikes/spike-like events.

```
#include <modelSpec.h>
```

Public Member Functions

- [weightUpdateModel](#) ()

Public Attributes

- string [simCode](#)
- string [simCodeEvt](#)

- string [simLearnPost](#)
- string [evntThreshold](#)
- string [synapseDynamics](#)
- vector< string > [varNames](#)
- vector< string > [varTypes](#)
- vector< string > [pNames](#)
- vector< string > [dpNames](#)

Names of dependent parameters of the model. These are assumed to be always of type "float".

- vector< string > [extraGlobalSynapseKernelParameters](#)

Additional parameter in the neuron kernel; it is translated to a population specific name but otherwise assumed to be one parameter per population rather than per synapse.

- vector< string > [extraGlobalSynapseKernelParameterTypes](#)

Additional parameters in the neuron kernel; they are translated to a population specific name but otherwise assumed to be one parameter per population rather than per synapse.

- [dpclass](#) * [dps](#)
- bool [needPreSt](#)
- bool [needPostSt](#)

10.21.1 Detailed Description

Structure to hold the information that defines a weightupdate model (a model of how spikes affect synaptic (and/or) (mostly) post-synaptic neuron variables. It also allows to define changes in response to post-synaptic spikes/spike-like events.

10.21.2 Constructor & Destructor Documentation

10.21.2.1 `weightUpdateModel::weightUpdateModel () [inline]`

10.21.3 Member Data Documentation

10.21.3.1 `vector<string> weightUpdateModel::dpNames`

Names of dependent parameters of the model. These are assumed to be always of type "float".

10.21.3.2 `dpclass* weightUpdateModel::dps`

10.21.3.3 `string weightUpdateModel::evntThreshold`

10.21.3.4 `vector<string> weightUpdateModel::extraGlobalSynapseKernelParameters`

Additional parameter in the neuron kernel; it is translated to a population specific name but otherwise assumed to be one parameter per population rather than per synapse.

10.21.3.5 `vector<string> weightUpdateModel::extraGlobalSynapseKernelParameterTypes`

Additional parameters in the neuron kernel; they are translated to a population specific name but otherwise assumed to be one parameter per population rather than per synapse.

10.21.3.6 `bool weightUpdateModel::needPostSt`

10.21.3.7 `bool weightUpdateModel::needPreSt`

10.21.3.8 `vector<string> weightUpdateModel::pNames`

10.21.3.9 `string weightUpdateModel::simCode`

10.21.3.10 `string weightUpdateModel::simCodeEvt`

10.21.3.11 `string weightUpdateModel::simLearnPost`

10.21.3.12 `string weightUpdateModel::synapseDynamics`

10.21.3.13 `vector<string> weightUpdateModel::varNames`

10.21.3.14 `vector<string> weightUpdateModel::varTypes`

The documentation for this class was generated from the following file:

- `lib/include/modelSpec.h`

Chapter 11

File Documentation

11.1 doxygen/00_MainPage.dox File Reference

11.2 doxygen/01_Installation.dox File Reference

11.3 doxygen/02_Quickstart.dox File Reference

11.4 doxygen/03_Examples.dox File Reference

11.5 doxygen/09_ReleaseNotes_v2.dox File Reference

11.6 doxygen/10_UserManual.dox File Reference

11.7 lib/include/ensureFtype.h File Reference

```
#include <string>
```

Functions

- void [doFinal](#) (string &code, unsigned int i, string type, unsigned int &state)
- string [ensureFtype](#) (string oldcode, string type)

Variables

- string [digits](#) = string("0123456789")
Function for converting code to contain only explicit single precision (float) constants.
- string [op](#) = string("+*/(<>= ,;")+string("\n")+string("\t")

11.7.1 Function Documentation

11.7.1.1 void doFinal (string & code, unsigned int i, string type, unsigned int & state)

11.7.1.2 `string ensureFType (string oldcode, string type)`

11.7.2 Variable Documentation

11.7.2.1 `string digits = string("0123456789")`

Function for converting code to contain only explicit single precision (float) constants.

11.7.2.2 `string op = string("+*/(<>=,;")+string("\n")+string("\t")`

11.8 lib/include/extra_neurons.h File Reference

Functions

- `n varNames clear ()`
- `n varNames push_back (tS("V"))`
- `n varTypes push_back (tS("float"))`
- `n varNames push_back (tS("V_NB"))`
- `n varNames push_back (tS("tSpike_NB"))`
- `n varNames push_back (tS("__regime_val"))`
- `n varTypes push_back (tS("int"))`
- `n pNames push_back (tS("VReset_NB"))`
- `n pNames push_back (tS("VThresh_NB"))`
- `n pNames push_back (tS("tRefrac_NB"))`
- `n pNames push_back (tS("VRest_NB"))`
- `n pNames push_back (tS("TAUm_NB"))`
- `n pNames push_back (tS("Cm_NB"))`
- `nModels push_back (n)`
- `n varNames push_back (tS("count_t_NB"))`
- `n pNames push_back (tS("max_t_NB"))`

Variables

- `n simCode`

11.8.1 Function Documentation

11.8.1.1 `ps dpNames clear ()`

11.8.1.2 `n varNames push_back (tS("V"))`

11.8.1.3 `ps varTypes push_back (tS("float"))`

11.8.1.4 `n varNames push_back (tS("V_NB"))`

11.8.1.5 `n varNames push_back (tS("tSpike_NB"))`

11.8.1.6 `n varNames push_back (tS("__regime_val"))`

11.8.1.7 `n varTypes push_back (tS("int"))`

11.8.1.8 `n pNames push_back (tS("VReset_NB"))`

11.8.1.9 `n pNames push_back (tS("VThresh_NB"))`

11.8.1.10 `n pNames push_back (tS("tRefrac_NB"))`

11.8.1.11 `n pNames push_back (tS("VRest_NB"))`

11.8.1.12 `n pNames push_back (tS("TAUm_NB"))`

11.8.1.13 `n pNames push_back (tS("Cm_NB"))`

11.8.1.14 `nModels push_back (n)`

11.8.1.15 `n varNames push_back (tS("count_t_NB"))`

11.8.1.16 `n pNames push_back (tS("max_t_NB"))`

11.8.2 Variable Documentation

11.8.2.1 `n simCode`

Initial value:

```
= tS(" \
    $(V) = -1000000; \
    if ($(__regime_val)==1) { \n \
$(V_NB) += (Isyn_NB/$ (Cm_NB) + ($ (VRest_NB) - $ (V_NB)) / $ (TAUm_NB)) *DT; \n \
    if ($ (V_NB) > $ (VThresh_NB)) { \n \
$(V_NB) = $ (VReset_NB); \n \
$(tSpike_NB) = t; \n \
    $(V) = 100000; \
    $(__regime_val) = 2; \n \
} \n \
} \n \
if ($(__regime_val)==2) { \n \
if (t-$(tSpike_NB) > $ (tRefrac_NB)) { \n \
    $(__regime_val) = 1; \n \
} \n \
} \n \
")
```

11.9 lib/include/extra_postsynapses.h File Reference

Functions

- `ps varNames clear ()`
- `ps varNames push_back (tS("g_PS"))`
- `ps varTypes push_back (tS("float"))`
- `ps pNames push_back (tS("tau_syn_PS"))`
- `ps pNames push_back (tS("E_PS"))`
- `postSynModels push_back (ps)`

Variables

- `ps postSyntoCurrent`
- `ps postSynDecay`

11.9.1 Function Documentation

11.9.1.1 `ps varNames clear ()`

11.9.1.2 `ps varNames push_back (tS("g_PS"))`

11.9.1.3 `ps varTypes push_back (tS("float"))`

11.9.1.4 `ps pNames push_back (tS("tau_syn_PS"))`

11.9.1.5 `ps pNames push_back (tS("E_PS"))`

11.9.1.6 `postSynModels push_back (ps)`

11.9.2 Variable Documentation

11.9.2.1 `ps postSynDecay`

Initial value:

```
= tS(" \
    $(g_PS) += (-$(g_PS)/$(tau_syn_PS))*DT; \n \
    $(inSyn) = 0; \
")
```

11.9.2.2 `ps postSyntoCurrent`

Initial value:

```
= tS(" \
0; \n \
    float Isyn_NB = 0; \n \
    { \n \
        float v_PS = lV_NB; \n \
        float g_in_PS = $(inSyn); \
    $(g_PS) = $(g_PS)+g_in_PS; \n \
    Isyn_NB += ($(g_PS)*$(E_PS)-v_PS)); \n \
    } \n \
")
```

11.10 SpineML_to_GeNN/extra_postsynapses.h File Reference

Functions

- `ps varNames clear ()`
- `ps varNames push_back (tS("g_PS"))`
- `ps varTypes push_back (tS("float"))`
- `ps pNames push_back (tS("tau_syn_PS"))`
- `ps pNames push_back (tS("E_PS"))`
- `postSynModels push_back (ps)`

Variables

- `ps postSyntoCurrent`
- `ps postSynDecay`

11.10.1 Function Documentation

11.10.1.1 `ps varNames clear ()`

11.10.1.2 `ps varNames push_back (tS("g_PS"))`

11.10.1.3 `ps varTypes push_back (tS("float"))`

11.10.1.4 `ps pNames push_back (tS("tau_syn_PS"))`

11.10.1.5 `ps pNames push_back (tS("E_PS"))`

11.10.1.6 `postSynModels push_back (ps)`

11.10.2 Variable Documentation

11.10.2.1 `ps postSynDecay`

Initial value:

```
= tS(" \
    $(g_PS) += (-$(g_PS)/$(tau_syn_PS))*DT; \n \
    $(inSyn) = 0; \
")
```

11.10.2.2 `ps postSyntoCurrent`

Initial value:

```
= tS(" \
0; \n \
    float Isyn_NB = 0; \n \
    { \n \
        float v_PS = lV_NB; \n \
        float g_in_PS = $(inSyn); \
    $(g_PS) = $(g_PS)+g_in_PS; \n \
    Isyn_NB += ($(g_PS)*$(E_PS)-v_PS)); \n \
    } \n \
")
```

11.11 lib/include/extra_weightupdates.h File Reference

11.12 lib/include/global.h File Reference

Global header file containing a few global variables. Part of the code generation section.

```
#include <iostream>
#include <cstring>
#include <string>
#include <sstream>
#include <vector>
#include <cmath>
#include <cuda_runtime.h>
#include "toString.h"
#include <stdint.h>
```

Macros

- `#define _GLOBAL_H_`
macro for avoiding multiple inclusion during compilation

Variables

- `int neuronBlkSz`
- `int synapseBlkSz`
- `int learnBlkSz`
- `cudaDeviceProp * deviceProp`
- `int theDev`
- `int hostCount`
Global variable containing the number of hosts within the local compute cluster.
- `int deviceCount`
Global variable containing the number of CUDA devices found on this host.
- `int optimiseBlockSize = 1`
Flag for signalling whether or not block size optimisation should be performed.
- `int UIntSz = sizeof(unsigned int) * 8`
size of the unsigned int variable type on the local architecture
- `int logUIntSz = (int) (logf((float) UIntSz) / logf(2.0f) + 1e-5f)`
logarithm of the size of the unsigned int variable type on the local architecture

11.12.1 Detailed Description

Global header file containing a few global variables. Part of the code generation section.

This global header file also takes care of including some generally used cuda support header files.

11.12.2 Macro Definition Documentation

11.12.2.1 `#define _GLOBAL_H_`

macro for avoiding multiple inclusion during compilation

11.12.3 Variable Documentation

11.12.3.1 `int deviceCount`

Global variable containing the number of CUDA devices found on this host.

11.12.3.2 `cudaDeviceProp* deviceProp`

11.12.3.3 `int hostCount`

Global variable containing the number of hosts within the local compute cluster.

11.12.3.4 `int learnBlkSz`

11.12.3.5 `int logUIntSz = (int) (logf((float) UIntSz) / logf(2.0f) + 1e-5f)`

logarithm of the size of the unsigned int variable type on the local architecture

11.12.3.6 int neuronBlkSz

11.12.3.7 int optimiseBlockSize = 1

Flag for signalling whether or not block size optimisation should be performed.

11.12.3.8 int synapseBlkSz

11.12.3.9 int theDev

11.12.3.10 int UIntSz = sizeof(unsigned int) * 8

size of the unsigned int variable type on the local architecture

11.13 lib/include/hr_time.cpp File Reference

This file contains the implementation of the [CStopWatch](#) class that provides a simple timing tool based on the system clock.

```
#include <cstdio>
#include "hr_time.h"
```

11.13.1 Detailed Description

This file contains the implementation of the [CStopWatch](#) class that provides a simple timing tool based on the system clock.

11.14 lib/include/hr_time.h File Reference

This header file contains the definition of the [CStopWatch](#) class that implements a simple timing tool using the system clock.

```
#include <sys/time.h>
```

Classes

- struct [stopWatch](#)
- class [CStopWatch](#)

11.14.1 Detailed Description

This header file contains the definition of the [CStopWatch](#) class that implements a simple timing tool using the system clock.

11.15 lib/include/modelSpec.cc File Reference

```
#include "utils.h"
```

Macros

- `#define _MODELSPEC_CC_`
macro for avoiding multiple inclusion during compilation

Functions

- void `initGeNN` ()

11.15.1 Macro Definition Documentation

11.15.1.1 `#define _MODELSPEC_CC_`

macro for avoiding multiple inclusion during compilation

11.15.2 Function Documentation

11.15.2.1 void `initGeNN` ()

11.16 lib/include/modelSpec.h File Reference

Header file that contains the class (struct) definition of `neuronModel` for defining a neuron model and the class definition of `NNmodel` for defining a neuronal network model. Part of the code generation and generated code sections.

```
#include <vector>
#include "global.h"
```

Classes

- class `dpclass`
- struct `neuronModel`
class (struct) for specifying a neuron model.
- struct `postSynModel`
Structure to hold the information that defines a post-synaptic model (a model of how synapses affect post-synaptic neuron variables, classically in the form of a synaptic current). It also allows to define an equation for the dynamics that can be applied to the summed synaptic input variable "insyn".
- class `weightUpdateModel`
Structure to hold the information that defines a weightupdate model (a model of how spikes affect synaptic (and/or) (mostly) post-synaptic neuron variables. It also allows to define changes in response to post-synaptic spikes/spike-like events.
- class `NNmodel`
Structure to hold the information that defines synapse dynamics (a model of how synapse variables change over time, independent of or in addition to changes when spikes occur).

Macros

- `#define _MODELSPEC_H_`
macro for avoiding multiple inclusion during compilation
- `#define MAXNRN 6`
- `#define SYNTYPENO 4`

- `#define NOINP 0`
Macro attaching the name *NOINP* (no input) to 0.
- `#define CONSTINP 1`
Macro attaching the name *CONSTINP* (constant input) to 1.
- `#define MATINP 2`
Macro attaching the name *MATINP* (explicit input defined as a matrix) to 2.
- `#define INPRULE 3`
Macro attaching the name *INPRULE* (explicit dynamic input defined as a rule) to 3.
- `#define RANDNINP 4`
Macro attaching the name *RANDNINP* (Random input with Gaussian distribution, calculated real time on the device by the generated code) to 4 (TODO, not implemented yet)
- `#define ALLTOALL 0`
Macro attaching the label "ALLTOALL" to connectivity type 0.
- `#define DENSE 1`
Macro attaching the label "DENSE" to connectivity type 1.
- `#define SPARSE 2`
Macro attaching the label "SPARSE" to connectivity type 2.
- `#define INDIVIDUALG 0`
Macro attaching the label "INDIVIDUALG" to method 0 for the definition of synaptic conductances.
- `#define GLOBALG 1`
Macro attaching the label "GLOBALG" to method 1 for the definition of synaptic conductances.
- `#define INDIVIDUALID 2`
Macro attaching the label "INDIVIDUALID" to method 2 for the definition of synaptic conductances.
- `#define NO_DELAY 1`
Macro used to indicate no synapse delay for the group (only one queue slot will be generated)
- `#define NOLEARNING 0`
Macro attaching the label "NOLEARNING" to flag 0.
- `#define LEARNING 1`
Macro attaching the label "LEARNING" to flag 1.
- `#define EXITSYN 0`
Macro attaching the label "EXITSYN" to flag 0 (excitatory synapse)
- `#define INHIBSYN 1`
Macro attaching the label "INHIBSYN" to flag 1 (inhibitory synapse)
- `#define TRUE 1`
Macro attaching the label "TRUE" to value 1.
- `#define FALSE 0`
Macro attaching the label "FALSE" to value 0.
- `#define CPU 0`
Macro attaching the label "CPU" to flag 0.
- `#define GPU 1`
Macro attaching the label "GPU" to flag 1.
- `#define FLOAT 0`
Macro attaching the label "FLOAT" to flag 0. Used by *NNModel::setPrecision()*
- `#define DOUBLE 1`
Macro attaching the label "DOUBLE" to flag 1. Used by *NNModel::setPrecision()*
- `#define AUTODEVICE -1`
Macro attaching the label *AUTODEVICE* to flag -1. Used by *setGPUDevice*.
- `#define SPK_THRESH_STDP 0.0f`
Macro defining the spiking threshold for the purposes of *STDP*.
- `#define MAXPOSTSYN 2`

Functions

- void [initGeNN](#) ()

Variables

- unsigned int [GeNNReady](#) = 0
- unsigned int [MAPNEURON](#)
variable attaching the name "MAPNEURON"
- unsigned int [POISSONNEURON](#)
variable attaching the name "POISSONNEURON"
- unsigned int [TRAUBMILES_FAST](#)
varianle attaching the name "TRAUBMILES_FAST"
- unsigned int [TRAUBMILES_ALTERNATIVE](#)
varianle attaching the name "TRAUBMILES_ALTERNATIVE"
- unsigned int [TRAUBMILES_SAFE](#)
varianle attaching the name "TRAUBMILES_SAFE"
- unsigned int [TRAUBMILES](#)
varianle attaching the name "TRAUBMILES"
- unsigned int [IZHIKEVICH](#)
variable attaching the name "IZHIKEVICH"
- unsigned int [IZHIKEVICH_V](#)
variable attaching the name "IZHIKEVICH_V"
- unsigned int [NSYNAPSE](#)
Variable attaching the name NSYNAPSE to predefined synapse type 0, which is a non-learning synapse.
- unsigned int [NGRADSYNAPSE](#)
Variable attaching the name NGRADSYNAPSE to predefined synapse type 1 which is a graded synapse wrt the presynaptic voltage.
- unsigned int [LEARN1SYNAPSE](#)
Variable attaching the name LEARN1SYNAPSE to the predefined synapse type 2 which is a learning using spike timing; uses a primitive STDP rule for learning.
- unsigned int [EXPDECAY](#)
- unsigned int [IZHIKEVICH_PS](#)

11.16.1 Detailed Description

Header file that contains the class (struct) definition of [neuronModel](#) for defining a neuron model and the class definition of [NNmodel](#) for defining a neuronal network model. Part of the code generation and generated code sections.

11.16.2 Macro Definition Documentation

11.16.2.1 `#define _MODELSPEC_H_`

macro for avoiding multiple inclusion during compilation

11.16.2.2 `#define ALLTOALL 0`

Macro attaching the label "ALLTOALL" to connectivity type 0.

11.16.2.3 `#define AUTODEVICE -1`

Macro attaching the label AUTODEVICE to flag -1. Used by setGPUDevice.

11.16.2.4 `#define CONSTINP 1`

Macro attaching the name CONSTINP (constant input) to 1.

11.16.2.5 `#define CPU 0`

Macro attaching the label "CPU" to flag 0.

11.16.2.6 `#define DENSE 1`

Macro attaching the label "DENSE" to connectivity type 1.

11.16.2.7 `#define DOUBLE 1`

Macro attaching the label "DOUBLE" to flag 1. Used by NNModel::setPrecision()

11.16.2.8 `#define EXITSYN 0`

Macro attaching the label "EXITSYN" to flag 0 (excitatory synapse)

11.16.2.9 `#define FALSE 0`

Macro attaching the label "FALSE" to value 0.

11.16.2.10 `#define FLOAT 0`

Macro attaching the label "FLOAT" to flag 0. Used by NNModel::setPrecision()

11.16.2.11 `#define GLOBALG 1`

Macro attaching the label "GLOBALG" to method 1 for the definition of synaptic conductances.

11.16.2.12 `#define GPU 1`

Macro attaching the label "GPU" to flag 1.

11.16.2.13 `#define INDIVIDUALG 0`

Macro attaching the label "INDIVIDUALG" to method 0 for the definition of synaptic conductances.

11.16.2.14 `#define INDIVIDUALID 2`

Macro attaching the label "INDIVIDUALID" to method 2 for the definition of synaptic conductances.

11.16.2.15 #define INHIBSYN 1

Macro attaching the label "INHIBSYN" to flag 1 (inhibitory synapse)

11.16.2.16 #define INPRULE 3

Macro attaching the name INPRULE (explicit dynamic input defined as a rule) to 3.

11.16.2.17 #define LEARNING 1

Macro attaching the label "LEARNING" to flag 1.

11.16.2.18 #define MATINP 2

Macro attaching the name MATINP (explicit input defined as a matrix) to 2.

11.16.2.19 #define MAXNRN 6**11.16.2.20 #define MAXPOSTSYN 2****11.16.2.21 #define NO_DELAY 1**

Macro used to indicate no synapse delay for the group (only one queue slot will be generated)

11.16.2.22 #define NOINP 0

Macro attaching the name NOINP (no input) to 0.

11.16.2.23 #define NOLEARNING 0

Macro attaching the label "NOLEARNING" to flag 0.

11.16.2.24 #define RANDNINP 4

Macro attaching the name RANDNINP (Random input with Gaussian distribution, calculated real time on the device by the generated code) to 4 (TODO, not implemented yet)

11.16.2.25 #define SPARSE 2

Macro attaching the label "SPARSE" to connectivity type 2.

11.16.2.26 #define SPK_THRESH_STDP 0.0f

Macro defining the spiking threshold for the purposes of STDP.

11.16.2.27 #define SYNTYPENO 4**11.16.2.28 #define TRUE 1**

Macro attaching the label "TRUE" to value 1.

11.16.3 Function Documentation

11.16.3.1 void initGeNN ()

11.16.4 Variable Documentation

11.16.4.1 unsigned int EXPDECAY

11.16.4.2 unsigned int GeNNReady = 0

11.16.4.3 unsigned int IZHIKEVICH

variable attaching the name "IZHIKEVICH"

11.16.4.4 unsigned int IZHIKEVICH_PS

11.16.4.5 unsigned int IZHIKEVICH_V

variable attaching the name "IZHIKEVICH_V"

11.16.4.6 unsigned int LEARN1SYNAPSE

Variable attaching the name LEARN1SYNAPSE to the predefined synapse type 2 which is a learning using spike timing; uses a primitive STDP rule for learning.

11.16.4.7 unsigned int MAPNEURON

variable attaching the name "MAPNEURON"

11.16.4.8 unsigned int NGRADSYNAPSE

Variable attaching the name NGRADSYNAPSE to predefined synapse type 1 which is a graded synapse wrt the presynaptic voltage.

11.16.4.9 unsigned int NSYNAPSE

Variable attaching the name NSYNAPSE to predefined synapse type 0, which is a non-learning synapse.

11.16.4.10 unsigned int POISSONNEURON

variable attaching the name "POISSONNEURON"

11.16.4.11 unsigned int TRAUBMILES

varianle attaching the name "TRAUBMILES"

11.16.4.12 unsigned int TRAUBMILES_ALTERNATIVE

varianle attaching the name "TRAUBMILES_ALTERNATIVE"

11.16.4.13 unsigned int TRAUBMILES_FAST

varianle attaching the name "TRAUBMILES_FAST"

11.16.4.14 unsigned int TRAUBMILES_SAFE

varianle attaching the name "TRAUBMILES_SAFE"

11.17 lib/include/numlib/gauss.cc File Reference

Contains the implementation of the Gaussian random number generator class [randomGauss](#).

```
#include "gauss.h"
```

Macros

- `#define GAUSS_CC`
macro for avoiding multiple inclusion during compilation

11.17.1 Detailed Description

Contains the implementation of the Gaussian random number generator class [randomGauss](#).

11.17.2 Macro Definition Documentation

11.17.2.1 #define GAUSS_CC

macro for avoiding multiple inclusion during compilation

11.18 lib/include/numlib/gauss.h File Reference

Random number generator for Gaussian random variable with mean 0 and standard deviation 1.

```
#include <cmath>
#include "randomGen.h"
#include "randomGen.cc"
#include "gauss.cc"
```

Classes

- class [randomGauss](#)
Class random Gauss encapsulates the methods for generating random neumbers with Gaussian distribution.

Macros

- `#define GAUSS_H`
macro for avoiding multiple inclusion during compilation

11.18.1 Detailed Description

Random number generator for Gaussian random variable with mean 0 and standard deviation 1.

This random number generator is based on the ratio of uniforms method by A.J. Kinderman and J.F. Monahan and improved with quadratic bounding curves by J.L. Leva. Taken from Algorithm 712 ACM Trans. Math. Softw. 18 p. 454. (the necessary uniform random variables are obtained from the ISAAC random number generator; C++ Implementation by Quinn Tyler Jackson of the RG invented by Bob Jenkins Jr.).

11.18.2 Macro Definition Documentation

11.18.2.1 `#define GAUSS_H`

macro for avoiding multiple inclusion during compilation

11.19 lib/include/numlib/randomGen.cc File Reference

Contains the implementation of the ISAAC random number generator class for uniformly distributed random numbers and for a standard random number generator based on the C function `rand()`.

```
#include "randomGen.h"
```

Macros

- `#define RANDOMGEN_CC`
macro for avoiding multiple inclusion during compilation

11.19.1 Detailed Description

Contains the implementation of the ISAAC random number generator class for uniformly distributed random numbers and for a standard random number generator based on the C function `rand()`.

11.19.2 Macro Definition Documentation

11.19.2.1 `#define RANDOMGEN_CC`

macro for avoiding multiple inclusion during compilation

11.20 lib/include/numlib/randomGen.h File Reference

header file containing the class definition for a uniform random generator based on the ISAAC random number generator

```
#include <time.h>
#include <limits.h>
#include <stdlib.h>
#include "isaac.hpp"
#include <assert.h>
```

Classes

- class [randomGen](#)
Class [randomGen](#) which implements the ISAAC random number generator for uniformly distributed random numbers.
- class [stdRG](#)

Macros

- `#define` [RANDOMGEN_H](#)
macro for avoiding multiple inclusion during compilation

11.20.1 Detailed Description

header file containing the class definition for a uniform random generator based on the ISAAC random number generator

11.20.2 Macro Definition Documentation

11.20.2.1 `#define` [RANDOMGEN_H](#)

macro for avoiding multiple inclusion during compilation

11.21 lib/include/numlib/simpleBit.h File Reference

Contains three macros that allow simple bit manipulations on an (presumably unsigned) 64 bit integer.

```
#include <cassert>
#include <cmath>
```

Macros

- `#define` [SIMPLEBIT_H](#)
macro for avoiding multiple inclusion during compilation
- `#define` [B](#)(*x*, *i*) ((*x*) & (0x80000000 >> (*i*)))
Extract the bit at the specified position *i* from *x*.
- `#define` [setB](#)(*x*, *i*) *x* = ((*x*) | (0x80000000 >> (*i*)))
Set the bit at the specified position *i* in *x* to 1.
- `#define` [delB](#)(*x*, *i*) *x* = ((*x*) & (~ (0x80000000 >> (*i*))))
Set the bit at the specified position *i* in *x* to 0.

11.21.1 Detailed Description

Contains three macros that allow simple bit manipulations on an (presumably unsigned) 64 bit integer.

11.21.2 Macro Definition Documentation

11.21.2.1 `#define` [B](#)(*x*, *i*) ((*x*) & (0x80000000 >> (*i*)))

Extract the bit at the specified position *i* from *x*.

11.21.2.2 `#define delB(x, i) x= ((x) & (~ (0x80000000 >> (i))))`

Set the bit at the specified position i in x to 0.

11.21.2.3 `#define setB(x, i) x= ((x) | (0x80000000 >> (i)))`

Set the bit at the specified position i in x to 1.

11.21.2.4 `#define SIMPLEBIT_H`

macro for avoiding multiple inclusion during compilation

11.22 lib/include/sparseUtils.cc File Reference

```
#include <stdio.h>
#include <math.h>
```

Macros

- `#define sparse_utils_cc`

Functions

- `template<class DATATYPE >`
`unsigned int countEntriesAbove (DATATYPE *Array, int sz, DATATYPE includeAbove)`
- `template<class DATATYPE >`
`DATATYPE getG (DATATYPE *wuvar, Conductance *sparseStruct, int x, int y)`
- `template<class DATATYPE >`
`float getSparseVar (DATATYPE *wuvar, Conductance *sparseStruct, int x, int y)`
- `template<class DATATYPE >`
`void setSparseConnectivityFromDense (DATATYPE *wuvar, int preN, int postN, DATATYPE *tmp_gRNPN, Conductance *sparseStruct)`
- `template<class DATATYPE >`
`void createSparseConnectivityFromDense (DATATYPE *wuvar, int preN, int postN, DATATYPE *tmp_gRNPN, Conductance *sparseStruct, bool runTest)`
- `void createPosttoPreArray (unsigned int preN, unsigned int postN, Conductance *sparseStruct)`
- `void strsearch (string &s, const string trg)`

!!!find var to check if a string is used in a code (atm it is used to create post-to-pre arrays)

11.22.1 Macro Definition Documentation

11.22.1.1 `#define sparse_utils_cc`

11.22.2 Function Documentation

11.22.2.1 `template<class DATATYPE > unsigned int countEntriesAbove (DATATYPE * Array, int sz, DATATYPE includeAbove)`

11.22.2.2 `void createPosttoPreArray (unsigned int preN, unsigned int postN, Conductance * sparseStruct)`

- 11.22.2.3 `template<class DATATYPE > void createSparseConnectivityFromDense (DATATYPE * wuvar, int preN, int postN, DATATYPE * tmp_gRNPN, Conductance * sparseStruct, bool runTest)`
- 11.22.2.4 `template<class DATATYPE > DATATYPE getG (DATATYPE * wuvar, Conductance * sparseStruct, int x, int y)`
- 11.22.2.5 `template<class DATATYPE > float getSparseVar (DATATYPE * wuvar, Conductance * sparseStruct, int x, int y)`
- 11.22.2.6 `template<class DATATYPE > void setSparseConnectivityFromDense (DATATYPE * wuvar, int preN, int postN, DATATYPE * tmp_gRNPN, Conductance * sparseStruct)`
- 11.22.2.7 `void strsearch (string & s, const string trg)`

!!!!find var to check if a string is used in a code (atm it is used to create post-to-pre arrays)

11.23 lib/include/testi.cc File Reference

```
#include <iostream>
#include <string>
#include "ensureFtype.h"
```

Functions

- int [main](#) ()

11.23.1 Function Documentation

11.23.1.1 `int main (void)`

11.24 lib/include/toString.h File Reference

Contains a template function for string conversion from `const char*` to C++ string.

```
#include <string>
#include <sstream>
```

Macros

- `#define _TOSTRING_H_`
macro for avoiding multiple inclusion during compilation
- `#define tS(X) toString(X)`
Macro providing the abbreviated syntax `tS()` instead of `toString()`.

Functions

- `template<typename T > std::string toString (T t)`
template function for string conversion from `const char` to C++ string*

11.24.1 Detailed Description

Contains a template function for string conversion from `const char*` to C++ string.

11.24.2 Macro Definition Documentation

11.24.2.1 `#define _TOSTRING_H_`

macro for avoiding multiple inclusion during compilation

11.24.2.2 `#define tS(X) toString(X)`

Macro providing the abbreviated syntax `tS()` instead of `toString()`.

11.24.3 Function Documentation

11.24.3.1 `template<typename T> std::string toString (T t)`

template function for string conversion from `const char*` to C++ string

11.25 lib/include/usertools.h File Reference

```
#include "toString.h"
```

11.26 lib/include/utils.h File Reference

This file contains standard utility functions provide within the NVIDIA CUDA software development toolkit (SDK). The remainder of the file contains a function that defines the standard neuron models.

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <memory>
#include <fstream>
#include <cmath>
#include <cuda_runtime.h>
#include "modelSpec.h"
#include "toString.h"
#include "ensureFtype.h"
#include "extra_neurons.h"
#include "extra_postsynapses.h"
#include "extra_weightupdates.h"
#include "numlib/simpleBit.h"
```

Classes

- class `rulkovdp`

Class defining the dependent parameters of teh Rulkov map neuron.

- class [expDecayDp](#)

Class defining the dependent parameter for exponential decay.

- class [pwSTDP](#)

TODO This class definition may be code-generated in a future release.

Macros

- `#define _UTILS_H_`

macro for avoiding multiple inclusion during compilation

- `#define CHECK_CUDA_ERRORS(call)`

Macro for wrapping cuda runtime function calls and catching any errors that may be thrown.

Functions

- void [writeHeader](#) (ostream &os)

Function to write the comment header denoting file authorship and contact details into the generated code.

- void [substitute](#) (string &s, const string trg, const string rep)

Tool for substituting strings in the neuron code strings or other templates.

- bool [find](#) (string &s, const string trg)

Tool for finding strings in another string.

- unsigned int [theSize](#) (string type)

Tool for determining the size of variable types on the current architecture.

- void [prepareStandardModels](#) ()

Function that defines standard neuron models.

- void [preparePostSynModels](#) ()

Function that prepares the standard post-synaptic models, including their variables, parameters, dependent parameters and code strings.

- void [prepareWeightUpdateModels](#) ()

Function that prepares the standard (pre) synaptic models, including their variables, parameters, dependent parameters and code strings.

Variables

- vector< [neuronModel](#) > [nModels](#)

Global C++ vector containing all neuron model descriptions.

- vector< [postSynModel](#) > [postSynModels](#)

Global C++ vector containing all post-synaptic update model descriptions.

- vector< [weightUpdateModel](#) > [weightUpdateModels](#)

Global C++ vector containing all weightupdate model descriptions.

11.26.1 Detailed Description

This file contains standard utility functions provide within the NVIDIA CUDA software development toolkit (SDK). The remainder of the file contains a function that defines the standard neuron models.

11.26.2 Macro Definition Documentation

11.26.2.1 `#define _UTILS_H_`

macro for avoiding multiple inclusion during compilation

11.26.2.2 #define CHECK_CUDA_ERRORS(*call*)**Value:**

```

{
    cudaError_t error = call;
    if (error != cudaSuccess)
    {
        fprintf(stderr, "%s: %i: cuda error %i: %s\n",
            __FILE__, __LINE__, (int)error, cudaGetErrorString(error));
        exit(EXIT_FAILURE);
    }
}

```

Macro for wrapping cuda runtime function calls and catching any errors that may be thrown.

11.26.3 Function Documentation**11.26.3.1 bool find (string & s, const string *trg*)**

Tool for finding strings in another string.

11.26.3.2 void preparePostSynModels ()

Function that prepares the standard post-synaptic models, including their variables, parameters, dependent parameters and code strings.

11.26.3.3 void prepareStandardModels ()

Function that defines standard neuron models.

The neuron models are defined and added to the C++ vector `nModels` that is holding all neuron model descriptions. User defined neuron models can be appended to this vector later in (a) separate function(s).

11.26.3.4 void prepareWeightUpdateModels ()

Function that prepares the standard (pre) synaptic models, including their variables, parameters, dependent parameters and code strings.

11.26.3.5 void substitute (string & s, const string *trg*, const string *rep*)

Tool for substituting strings in the neuron code strings or other templates.

11.26.3.6 unsigned int theSize (string *type*)

Tool for determining the size of variable types on the current architecture.

11.26.3.7 void writeHeader (ostream & os)

Function to write the comment header denoting file authorship and contact details into the generated code.

11.26.4 Variable Documentation**11.26.4.1 vector<neuronModel> nModels**

Global C++ vector containing all neuron model descriptions.

11.26.4.2 `vector<postSynModel> postSynModels`

Global C++ vector containing all post-synaptic update model descriptions.

11.26.4.3 `vector<weightUpdateModel> weightUpdateModels`

Global C++ vector containing all weightupdate model descriptions.

11.27 `lib/src/CodeHelper.cc` File Reference

```
#include <iostream>
#include <cstring>
#include <string>
#include <sstream>
#include <vector>
```

Classes

- class [CodeHelper](#)

Macros

- `#define __CODE_HELPER_CC`
- `#define SAVEP(X) "(" << X << ")"`
- `#define OB(X) hlp.openBrace(X)`
- `#define CB(X) hlp.closeBrace(X)`
- `#define ENDL hlp.endl()`

11.27.1 Macro Definition Documentation

11.27.1.1 `#define __CODE_HELPER_CC`

11.27.1.2 `#define CB(X) hlp.closeBrace(X)`

11.27.1.3 `#define ENDL hlp.endl()`

11.27.1.4 `#define OB(X) hlp.openBrace(X)`

11.27.1.5 `#define SAVEP(X) "(" << X << ")"`

11.28 `lib/src/generateALL.cc` File Reference

Main file combining the code for code generation. Part of the code generation section.

```
#include "global.h"
#include "modelSpec.h"
#include "modelSpec.cc"
#include "generateKernels.cc"
#include "generateRunner.cc"
#include "generateCPU.cc"
#include <sys/stat.h>
```

Functions

- void [generate_model_runner](#) ([NNmodel](#) &model, string path)
This function will call the necessary sub-functions to generate the code for simulating a model.
- int [chooseDevice](#) (ostream &mos, [NNmodel](#) *&model, string path)
Helper function that prepares data structures and detects the hardware properties to enable the code generation code that follows.
- int [main](#) (int argc, char *argv[])
Main entry point for the generateALL executable that generates the code for GPU and CPU.

11.28.1 Detailed Description

Main file combining the code for code generation. Part of the code generation section.

The file includes separate files for generating kernels ([generateKernels.cc](#)), generating the CPU side code for running simulations on either the CPU or GPU ([generateRunner.cc](#)) and for CPU-only simulation code ([generateCPU.cc](#)).

11.28.2 Function Documentation

11.28.2.1 int chooseDevice (ostream & mos, NNmodel *& model, string path)

Helper function that prepares data structures and detects the hardware properties to enable the code generation code that follows.

The main tasks in this function are the detection and characterization of the GPU device present (if any), choosing which GPU device to use, finding and appropriate block size, taking note of the major and minor version of the C↔UDA enabled device chosen for use, and populating the list of standard neuron models. The chosen device number is returned.

Parameters

<i>mos</i>	output stream for messages
<i>model</i>	the nn model we are generating code for
<i>path</i>	path the generated code will be deposited

11.28.2.2 void generate_model_runner (NNmodel & model, string path)

This function will call the necessary sub-functions to generate the code for simulating a model.

Parameters

<i>model</i>	Model description
<i>path</i>	Path where the generated code will be deposited

11.28.2.3 int main (int argc, char * argv[])

Main entry point for the generateALL executable that generates the code for GPU and CPU.

The main function is the entry point for the code generation engine. It prepares the system and then invokes [generate_model_runner](#) to initiate the different parts of actual code generation.

Parameters

<i>argc</i>	number of arguments; expected to be 2
<i>argv</i>	Arguments; expected to contain the target directory for code generation.

11.29 lib/src/generateCPU.cc File Reference

Functions for generating code that will run the neuron and synapse simulations on the CPU. Part of the code generation section.

```
#include <string>
#include "CodeHelper.cc"
```

Functions

- void [genNeuronFunction](#) ([NNmodel](#) &model, string &path, ostream &mos)
Function that generates the code of the function the will simulate all neurons on the CPU.
- void [generate_process_presynaptic_events_code_CPU](#) (ostream &os, [NNmodel](#) &model, unsigned int src, unsigned int trg, int i, string &localID, unsigned int inSynNo, string postfix)
Function for generating the CUDA synapse kernel code that handles presynaptic spikes or spike type events.
- void [genSynapseFunction](#) ([NNmodel](#) &model, string &path, ostream &mos)
Function that generates code that will simulate all synapses of the model on the CPU.

11.29.1 Detailed Description

Functions for generating code that will run the neuron and synapse simulations on the CPU. Part of the code generation section.

11.29.2 Function Documentation

11.29.2.1 void [generate_process_presynaptic_events_code_CPU](#) (ostream & *os*, [NNmodel](#) & *model*, unsigned int *src*, unsigned int *trg*, int *i*, string & *localID*, unsigned int *inSynNo*, string *postfix*)

Function for generating the CUDA synapse kernel code that handles presynaptic spikes or spike type events.

Parameters

<i>os</i>	output stream for code
<i>model</i>	the neuronal network model to generate code for
<i>src</i>	the number of the src neuron population
<i>trg</i>	the number of the target neuron population
<i>i</i>	the index of the synapse group being processed
<i>localID</i>	the variable name of the local ID of the thread within the synapse group
<i>inSynNo</i>	the ID number of the current synapse population as the incoming population to the target neuron population
<i>postfix</i>	whether to generate code for true spikes or spike type events

11.29.2.2 void [genNeuronFunction](#) ([NNmodel](#) & *model*, string & *path*, ostream & *mos*)

Function that generates the code of the function the will simulate all neurons on the CPU.

Parameters

<i>model</i>	Model description
<i>path</i>	output stream for code
<i>mos</i>	output stream for messages

11.29.2.3 void genSynapseFunction (NNmodel & *model*, string & *path*, ostream & *mos*)

Function that generates code that will simulate all synapses of the model on the CPU.

Parameters

<i>model</i>	Model description
<i>path</i>	Path for code generation
<i>mos</i>	output stream for messages

11.30 lib/src/generateKernels.cc File Reference

Contains functions that generate code for CUDA kernels. Part of the code generation section.

```
#include <string>
#include "global.h"
#include "CodeHelper.cc"
```

Functions

- void [name_substitutions](#) (string &code, string prefix, vector< string > &names, string postfix=string(""))
- void [value_substitutions](#) (string &code, vector< string > &names, vector< float > &values)
- void [extended_name_substitutions](#) (string &code, string prefix, vector< string > &names, string ext, string postfix=string(""))
- void [extended_value_substitutions](#) (string &code, vector< string > &names, string ext, vector< float > &values)
- void [genNeuronKernel](#) (NNmodel &model, string &path, ostream &mos)
- void [generate_process_presynaptic_events_code](#) (ostream &os, NNmodel &model, unsigned int src, unsigned int trg, int i, string &localID, unsigned int inSynNo, string postfix)

Function for generating the CUDA synapse kernel code that handles presynaptic spikes or spike type events.

- void [genSynapseKernel](#) (NNmodel &model, string &path, ostream &mos)

Function for generating a CUDA kernel for simulating all synapses.

Variables

- unsigned int [nt](#)

Function for generating the CUDA kernel that simulates all neurons in the model.

- short * [isGrpVarNeeded](#)
- [CodeHelper](#) [hlp](#)

11.30.1 Detailed Description

Contains functions that generate code for CUDA kernels. Part of the code generation section.

11.30.2 Function Documentation

11.30.2.1 `void extended_name_substitutions (string & code, string prefix, vector< string > & names, string ext, string postfix = string(""))`

11.30.2.2 `void extended_value_substitutions (string & code, vector< string > & names, string ext, vector< float > & values)`

11.30.2.3 `void generate_process_presynaptic_events_code (ostream & os, NNmodel & model, unsigned int src, unsigned int trg, int i, string & localID, unsigned int inSynNo, string postfix)`

Function for generating the CUDA synapse kernel code that handles presynaptic spikes or spike type events.

Parameters

<i>os</i>	output stream for code
<i>model</i>	the neuronal network model to generate code for
<i>src</i>	the number of the src neuron population
<i>trg</i>	the number of the target neuron population
<i>i</i>	the index of the synapse group being processed
<i>localID</i>	the variable name of the local ID of the thread within the synapse group
<i>inSynNo</i>	the ID number of the current synapse population as the incoming population to the target neuron population
<i>postfix</i>	whether to generate code for true spikes or spike type events

11.30.2.4 `void genNeuronKernel (NNmodel & model, string & path, ostream & mos)`

Binary flag for the sparse synapses to use atomic operations when the number of connections is bigger than the block size, and shared variables otherwise

Parameters

<i>model</i>	Model description
<i>path</i>	path for code output
<i>mos</i>	output stream for messages

11.30.2.5 `void genSynapseKernel (NNmodel & model, string & path, ostream & mos)`

Function for generating a CUDA kernel for simulating all synapses.

This functions generates code for global variables on the GPU side that are synapse-related and the actual CUDA kernel for simulating one time step of the synapses. < "id" if first synapse group, else "lid". lid =(thread index- last thread of the last synapse group)

Parameters

<i>model</i>	Model description
<i>path</i>	Path for code output
<i>mos</i>	output stream for messages

11.30.2.6 `void name_substitutions (string & code, string prefix, vector< string > & names, string postfix = string(""))`

11.30.2.7 `void value_substitutions (string & code, vector< string > & names, vector< float > & values)`

11.30.3 Variable Documentation

11.30.3.1 CodeHelper hlp

11.30.3.2 short* isGrpVarNeeded

11.30.3.3 unsigned int nt

Function for generating the CUDA kernel that simulates all neurons in the model.

The code generated upon execution of this function is for defining GPU side global variables that will hold model state in the GPU global memory and for the actual kernel function for simulating the neurons for one time step.

11.31 lib/src/generateRunner.cc File Reference

Contains functions to generate code for running the simulation on the GPU, and for I/O convenience functions between GPU and CPU space. Part of the code generation section.

Functions

- void [genRunner](#) ([NNmodel](#) &model, string path, ostream &mos)
A function that generates predominantly host-side code.
- void [genRunnerGPU](#) ([NNmodel](#) &model, string &path, ostream &mos)
A function to generate the code that simulates the model on the GPU.
- void [genRunnerCPU](#) ([NNmodel](#) &model, string &path, ostream &mos)
A function to generate code for an equivalent CPU-only simulation engine.

11.31.1 Detailed Description

Contains functions to generate code for running the simulation on the GPU, and for I/O convenience functions between GPU and CPU space. Part of the code generation section.

11.31.2 Function Documentation

11.31.2.1 void [genRunner](#) ([NNmodel](#) & *model*, string *path*, ostream & *mos*)

A function that generates predominantly host-side code.

In this function host-side functions and other code are generated, including: Global host variables, "allocatedMem()" function for allocating memories, "freeMem" function for freeing the allocated memories, "initialize" for initializing host variables, "gFunc" and "initGRaw()" for use with plastic synapses if such synapses exist in the model.

Parameters

<i>model</i>	Model description
<i>path</i>	path for code generation
<i>mos</i>	output stream for messages

11.31.2.2 void [genRunnerCPU](#) ([NNmodel](#) & *model*, string & *path*, ostream & *mos*)

A function to generate code for an equivalent CPU-only simulation engine.

The generated code provides the same functionality as the code generated for the GPU but only utilizing the C↔PU. That being so, no convenience functions for data transfer are necessary in this scenario and the only generated function is "stepTimeCPU()" for simulating a time step of the model on the CPU (using the CPU functions

for simulations, equivalent to the kernels in the GPU case, that are generated by `generateNeuronFunction()` and `generateSynapseFunction()`.

11.33 userproject/HHVclampGA_project/generate_run.cc File Reference

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <cstdlib>
#include <cmath>
#include <sys/stat.h>
#include "usertools.h"
```

Functions

- int [main](#) (int argc, char *argv[])
Main entry point for generate_run.

11.33.1 Function Documentation

11.33.1.1 int main (int *argc*, char * *argv*[])

Main entry point for generate_run.

11.34 userproject/lzh_sparse_project/generate_run.cc File Reference

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <cstdlib>
#include <cmath>
#include <sys/stat.h>
```

Functions

- template<typename T >
std::string [toString](#) (T t)
Template function for string conversion.
- unsigned int [openFileGetMax](#) (unsigned int *array, unsigned int size, string name)
- int [main](#) (int argc, char *argv[])
Main entry point for generate_run.

11.34.1 Function Documentation

11.34.1.1 int main (int *argc*, char * *argv*[])

Main entry point for generate_run.

11.34.1.2 `unsigned int openFileGetMax (unsigned int * array, unsigned int size, string name)`

11.34.1.3 `template<typename T> std::string toString (T t)`

Template function for string conversion.

11.35 userproject/MBody1_project/generate_run.cc File Reference

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <cstdlib>
#include <cmath>
#include <locale>
#include <sys/stat.h>
```

Macros

- `#define tS(X) toString(X)`
Macro providing the abbreviated syntax `tS()` instead of `toString()`.

Functions

- `template<typename T> std::string toString (T t)`
template function for string conversion from `const char*` to C++ string
- `string toUpper (string s)`
- `string toLower (string s)`
- `int main (int argc, char *argv[])`
Main entry point for `generate_run`.

11.35.1 Macro Definition Documentation

11.35.1.1 `#define tS(X) toString(X)`

Macro providing the abbreviated syntax `tS()` instead of `toString()`.

11.35.2 Function Documentation

11.35.2.1 `int main (int argc, char * argv[])`

Main entry point for `generate_run`.

11.35.2.2 `string toLower (string s)`

11.35.2.3 `template<typename T> std::string toString (T t)`

template function for string conversion from `const char*` to C++ string

11.35.2.4 string toUpper (string s)

11.36 userproject/MBody_userdef_project/generate_run.cc File Reference

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <cstdlib>
#include <cmath>
#include <sys/stat.h>
```

Functions

- `template<typename T>`
`std::string toString (T t)`
Template function for string conversion.
- `int main (int argc, char *argv[])`
Main entry point for generate_run.

11.36.1 Function Documentation

11.36.1.1 int main (int argc, char * argv[])

Main entry point for generate_run.

11.36.1.2 template<typename T> std::string toString (T t)

Template function for string conversion.

11.37 userproject/OneComp_project/generate_run.cc File Reference

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <cstdlib>
#include <cmath>
#include <sys/stat.h>
```

Functions

- `template<typename T>`
`std::string toString (T t)`
Template function for string conversion.
- `int main (int argc, char *argv[])`
Main entry point for generate_run.

11.37.1 Function Documentation

11.37.1.1 `int main (int argc, char * argv[])`

Main entry point for `generate_run`.

11.37.1.2 `template<typename T> std::string toString (T t)`

Template function for string conversion.

11.38 userproject/PoissonIzh_project/generate_run.cc File Reference

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <cstdlib>
#include <cmath>
#include <sys/stat.h>
```

Functions

- `template<typename T> std::string toString (T t)`
Template function for string conversion.
- `int main (int argc, char *argv[])`
Main entry point for `generate_run`.

11.38.1 Function Documentation

11.38.1.1 `int main (int argc, char * argv[])`

Main entry point for `generate_run`.

11.38.1.2 `template<typename T> std::string toString (T t)`

Template function for string conversion.

11.39 tmp/model/classol_sim.cu File Reference

```
#include "classol_sim.h"
#include "sparseUtils.cc"
```

Functions

- `int main (int argc, char *argv[])`
This function is the entry point for running the simulation of the MBody1 model network.

11.39.1 Function Documentation

11.39.1.1 `int main (int argc, char * argv[])`

This function is the entry point for running the simulation of the MBody1 model network.

11.40 `userproject/MBody1_project/model/classol_sim.cu` File Reference

```
#include "classol_sim.h"
```

Functions

- `int main (int argc, char *argv[])`

This function is the entry point for running the simulation of the MBody1 model network.

11.40.1 Function Documentation

11.40.1.1 `int main (int argc, char * argv[])`

This function is the entry point for running the simulation of the MBody1 model network.

11.41 `userproject/MBody1_project/modelcmp/classol_sim.cu` File Reference

```
#include "classol_sim.h"
```

Functions

- `int main (int argc, char *argv[])`

This function is the entry point for running the simulation of the MBody1 model network.

11.41.1 Function Documentation

11.41.1.1 `int main (int argc, char * argv[])`

This function is the entry point for running the simulation of the MBody1 model network.

11.42 `userproject/MBody_userdef_project/model/classol_sim.cu` File Reference

```
#include "classol_sim.h"
```

Functions

- `int main (int argc, char *argv[])`

This function is the entry point for running the simulation of the MBody1 model network.

11.42.1 Function Documentation

11.42.1.1 int main (int argc, char * argv[])

This function is the entry point for running the simulation of the MBody1 model network.

11.43 tmp/model/classol_sim.h File Reference

```
#include <cassert>
#include "hr_time.cpp"
#include "utils.h"
#include <cuda_runtime.h>
#include "map_classol.cc"
```

Macros

- `#define RAND(Y, X) Y = Y * 1103515245 + 12345; X = (unsigned int)(Y >> 16) & 32767`
- `#define DBG_SIZE 10000`
- `#define PATTERNNO 100`
- `#define INPUTBASERATE 17`
- `#define T_REPORT_TME 1000.0`
- `#define SYN_OUT_TME 2000.0`
- `#define PAT_TIME 100.0`
- `#define PATFTIME 1.0`
- `#define TOTAL_TME 5000.0`

Variables

- float `t` = 0.0f
- unsigned int `iT` = 0
- int `patSetTime`
- int `patFireTime`
- `CStopWatch` timer

11.43.1 Macro Definition Documentation

11.43.1.1 `#define DBG_SIZE 10000`

11.43.1.2 `#define INPUTBASERATE 17`

11.43.1.3 `#define PAT_TIME 100.0`

11.43.1.4 `#define PATFTIME 1.0`

11.43.1.5 `#define PATTERNNO 100`

11.43.1.6 `#define RAND(Y, X) Y = Y * 1103515245 + 12345; X = (unsigned int)(Y >> 16) & 32767`

11.43.1.7 `#define SYN_OUT_TME 2000.0`

11.43.1.8 `#define T_REPORT_TME 1000.0`

11.43.1.9 `#define TOTAL_TME 5000.0`

11.43.2 Variable Documentation

11.43.2.1 `unsigned int iT = 0`

11.43.2.2 `int patFireTime`

11.43.2.3 `int patSetTime`

11.43.2.4 `float t = 0.0f`

11.43.2.5 `CStopWatch timer`

11.44 `userproject/MBody1_project/model/classol_sim.h` File Reference

```
#include <cassert>
#include "hr_time.cpp"
#include "utils.h"
#include <cuda_runtime.h>
#include <cfloat>
#include "MBody1.cc"
#include "map_classol.cc"
```

Macros

- `#define TIMING`
- `#define MYRAND(Y, X) Y = Y * 1103515245 + 12345; X = (Y >> 16);`
- `#define DBG_SIZE 10000`
- `#define PATTERNNO 100`
- `#define T_REPORT_TME 10000.0`
- `#define SYN_OUT_TME 20000.0`
- `#define PAT_TIME 100.0`
- `#define PATFTIME 1.5`
- `#define TOTAL_TME 5000.0`

Variables

- `scalar t = 0.0f`
- `unsigned int iT = 0`
- `scalar InputBaseRate = 2e-04`
- `int patSetTime`
- `int patFireTime`
- `CStopWatch timer`

11.44.1 Macro Definition Documentation

11.44.1.1 `#define DBG_SIZE 10000`

11.44.1.2 `#define MYRAND(Y, X) Y = Y * 1103515245 + 12345; X = (Y >> 16);`

11.44.1.3 `#define PAT_TIME 100.0`

11.44.1.4 `#define PATFTIME 1.5`

11.44.1.5 `#define PATTERNNO 100`

11.44.1.6 `#define SYN_OUT_TME 20000.0`

11.44.1.7 `#define T_REPORT_TME 10000.0`

11.44.1.8 `#define TIMING`

11.44.1.9 `#define TOTAL_TME 5000.0`

11.44.2 Variable Documentation

11.44.2.1 `scalar InputBaseRate = 2e-04`

11.44.2.2 `unsigned int iT = 0`

11.44.2.3 `int patFireTime`

11.44.2.4 `int patSetTime`

11.44.2.5 `scalar t = 0.0f`

11.44.2.6 `CStopWatch timer`

11.45 userproject/MBody1_project/modelcmp/classol_sim.h File Reference

```
#include <cassert>
#include "hr_time.cpp"
#include "utils.h"
#include <cuda_runtime.h>
#include "map_classol.cc"
```

Macros

- `#define MYRAND(Y, X) Y = Y * 1103515245 + 12345; X= (Y >> 16);`
- `#define DBG_SIZE 10000`
- `#define PATTERNNO 100`
- `#define T_REPORT_TME 1000.0`
- `#define SYN_OUT_TME 20000.0`
- `#define PAT_TIME 100.0`
- `#define PATFTIME 1.5`
- `#define TOTAL_TME 30000.0`

Variables

- `float t = 0.0f`
- `unsigned int iT = 0`
- `float InputBaseRate = 2e-04`
- `int patSetTime`
- `int patFireTime`
- `CStopWatch timer`

11.45.1 Macro Definition Documentation

11.45.1.1 `#define DBG_SIZE 10000`

11.45.1.2 `#define MYRAND(Y, X) Y = Y * 1103515245 +12345; X= (Y >> 16);`

11.45.1.3 `#define PAT_TIME 100.0`

11.45.1.4 `#define PATFTIME 1.5`

11.45.1.5 `#define PATTERNNO 100`

11.45.1.6 `#define SYN_OUT_TME 20000.0`

11.45.1.7 `#define T_REPORT_TME 1000.0`

11.45.1.8 `#define TOTAL_TME 30000.0`

11.45.2 Variable Documentation

11.45.2.1 `float InputBaseRate = 2e-04`

11.45.2.2 `unsigned int iT = 0`

11.45.2.3 `int patFireTime`

11.45.2.4 `int patSetTime`

11.45.2.5 `float t = 0.0f`

11.45.2.6 `CStopWatch timer`

11.46 userproject/MBody_userdef_project/model/classol_sim.h File Reference

```
#include <cassert>
#include "hr_time.cpp"
#include "utils.h"
#include <cuda_runtime.h>
#include "map_classol.cc"
```

Macros

- `#define MYRAND(Y, X) Y = Y * 1103515245 +12345; X= (Y >> 16);`
- `#define DBG_SIZE 10000`
- `#define PATTERNNO 100`
- `#define T_REPORT_TME 10000.0`
- `#define SYN_OUT_TME 20000.0`
- `#define PAT_TIME 100.0`
- `#define PATFTIME 1.5`
- `#define TOTAL_TME 200000.0`

Variables

- float `t` = 0.0f
- unsigned int `iT` = 0
- float `InputBaseRate` = 2e-04
- int `patSetTime`
- int `patFireTime`
- `CStopWatch` timer

11.46.1 Macro Definition Documentation

11.46.1.1 `#define DBG_SIZE 10000`

11.46.1.2 `#define MYRAND(Y, X) Y = Y * 1103515245 + 12345; X= (Y >> 16);`

11.46.1.3 `#define PAT_TIME 100.0`

11.46.1.4 `#define PATFTIME 1.5`

11.46.1.5 `#define PATTERNNO 100`

11.46.1.6 `#define SYN_OUT_TME 20000.0`

11.46.1.7 `#define T_REPORT_TME 10000.0`

11.46.1.8 `#define TOTAL_TME 200000.0`

11.46.2 Variable Documentation

11.46.2.1 float `InputBaseRate` = 2e-04

11.46.2.2 unsigned int `iT` = 0

11.46.2.3 int `patFireTime`

11.46.2.4 int `patSetTime`

11.46.2.5 float `t` = 0.0f

11.46.2.6 `CStopWatch` timer

11.47 tmp/model/map_classol.cc File Reference

```
#include "map_classol.h"
#include "MBody_userdef_CODE/runner.cc"
```

Macros

- `#define _MAP_CLASSOL_CC_`
macro for avoiding multiple inclusion during compilation

11.47.1 Macro Definition Documentation

11.47.1.1 `#define _MAP_CLASSOL_CC_`

macro for avoiding multiple inclusion during compilation

11.48 userproject/MBody1_project/model/map_classol.cc File Reference

```
#include "map_classol.h"  
#include "MBody1_CODE/runner.cc"
```

Macros

- `#define _MAP_CLASSOL_CC_`
macro for avoiding multiple inclusion during compilation

11.48.1 Macro Definition Documentation

11.48.1.1 `#define _MAP_CLASSOL_CC_`

macro for avoiding multiple inclusion during compilation

11.49 userproject/MBody1_project/modelcmp/map_classol.cc File Reference

```
#include "map_classol.h"  
#include "MBody1_CODE/runner.cc"
```

Macros

- `#define _MAP_CLASSOL_CC_`
macro for avoiding multiple inclusion during compilation

11.49.1 Macro Definition Documentation

11.49.1.1 `#define _MAP_CLASSOL_CC_`

macro for avoiding multiple inclusion during compilation

11.50 userproject/MBody_userdef_project/model/map_classol.cc File Reference

```
#include "map_classol.h"  
#include "MBody_userdef_CODE/runner.cc"  
#include "sparseUtils.cc"
```

Macros

- `#define _MAP_CLASSOL_CC_`
macro for avoiding multiple inclusion during compilation

11.50.1 Macro Definition Documentation

11.50.1.1 `#define _MAP_CLASSOL_CC_`

macro for avoiding multiple inclusion during compilation

11.51 tmp/model/map_classol.h File Reference

```
#include "MBody_userdef.cc"
```

Classes

- class `classol`
This class cponains the methods for running the MBody1 example model.

11.52 userproject/MBody1_project/model/map_classol.h File Reference

Classes

- class `classol`
This class cponains the methods for running the MBody1 example model.

11.53 userproject/MBody1_project/modelcmp/map_classol.h File Reference

```
#include "MBody1.cc"
```

Classes

- class `classol`
This class cponains the methods for running the MBody1 example model.

11.54 userproject/MBody_userdef_project/model/map_classol.h File Reference

```
#include "MBody_userdef.cc"
```

Classes

- class `classol`
This class cponains the methods for running the MBody1 example model.

11.55 tmp/model/MBody_userdef.cc File Reference

```
#include "modelSpec.h"
#include "modelSpec.cc"
#include "../userproject/include/sizes.h"
```

Classes

- class [pwSTDP](#)

TODO This class definition may be code-generated in a future release.

Macros

- #define [DT](#) 0.5

This defines the global time step at which the simulation will run.

Functions

- void [modelDefinition](#) ([NNmodel](#) &model)

This function defines the MBody1 model with user defined synapses.

Variables

- float [myPOI_p](#) [4]
- float [myPOI_ini](#) [4]
- float [stdTM_p](#) [7]
- float [stdTM_ini](#) [4]
- float [myPNKC_p](#) [3]
- float [postExpPNKC](#) [2]
- float [myPNLHI_p](#) [3]
- float [postExpPNLHI](#) [2]
- float [myLHIKC_p](#) [4]
- float [gLHIKC](#) = 0.006
- float [postExpLHIKC](#) [2]
- float [myKCDN_p](#) [13]
- float [postExpKCDN](#) [2]
- float [myDNDN_p](#) [4]
- float [gDNDN](#) = 0.01
- float [postExpDNDN](#) [2]
- float [postSynV](#) [0]
- float [myKCDN_userdef_p](#) [11]
- float * [gpPNKC](#) = new float[_NAL*_NMB]

11.55.1 Macro Definition Documentation

11.55.1.1 #define DT 0.5

This defines the global time step at which the simulation will run.

11.55.2 Function Documentation

11.55.2.1 void modelDefinition (NNmodel & *model*)

This function defines the MBody1 model with user defined synapses.

11.55.3 Variable Documentation

11.55.3.1 float gDNDN = 0.01

11.55.3.2 float gLHIC = 0.006

11.55.3.3 float* gpPNKC = new float[_NAL*_NMB]

11.55.3.4 float myDNDN_p[4]

Initial value:

```
= {
  -92.0,
  -30.0,
   8.0,
  50.0
}
```

11.55.3.5 float myKCDN_p[13]

Initial value:

```
= {
  0.0,
 -20.0,
  5.0,
  25.0,
 100.0,
50000.0,
100000.0,
 100.0,
  0.06,
  0.03,
 33.33,
 10.0,
0.00006
}
```

11.55.3.6 float myKCDN_userdef_p[11]

Initial value:

```
= {
 -20.0,
 25.0,
 100.0,
50000.0,
100000.0,
 100.0,
  0.06,
  0.03,
 33.33,
 10.0,
0.00006
}
```

11.55.3.7 float myLHIKC_p[4]**Initial value:**

```
= {  
  -92.0,  
  -40.0,  
   3.0,  
  50.0  
}
```

11.55.3.8 float myPNKC_p[3]**Initial value:**

```
= {  
  5.0,  
 -20.0,  
  1.0  
}
```

11.55.3.9 float myPNLHI_p[3]**Initial value:**

```
= {  
  0.0,  
 -20.0,  
  1.0  
}
```

11.55.3.10 float myPOL_ini[4]**Initial value:**

```
= {  
 -60.0,  
  0,  
 -10.0,  
}
```

11.55.3.11 float myPOL_p[4]**Initial value:**

```
= {  
  0.1,  
  2.5,  
 20.0,  
 -60.0  
}
```

11.55.3.12 float postExpDNDN[2]**Initial value:**

```
= {  
  8.0,  
 -92.0  
}
```


11.55.3.13 float postExpKCDN[2]**Initial value:**

```
= {  
    5.0,  
    0.0  
}
```

11.55.3.14 float postExpLHIKC[2]**Initial value:**

```
= {  
    3.0,  
   -92.0  
}
```

11.55.3.15 float postExpPNKC[2]**Initial value:**

```
= {  
    1.0,  
    0.0  
}
```

11.55.3.16 float postExpPNLHI[2]**Initial value:**

```
= {  
    1.0,  
    0.0  
}
```

11.55.3.17 float postSynV[0]**Initial value:**

```
= {  
}
```

11.55.3.18 float stdTM_ini[4]**Initial value:**

```
= {  
   -60.0,  
   0.0529324,  
   0.3176767,  
   0.5961207  
}
```

11.55.3.19 float stdTM_p[7]

Initial value:

```
= {
  7.15,
  50.0,
  1.43,
  -95.0,
  0.02672,
  -63.563,
  0.143
}
```

11.56 userproject/MBody_userdef_project/model/MBody_userdef.cc File Reference

```
#include "modelSpec.h"
#include "modelSpec.cc"
#include "sizes.h"
```

Classes

- class [pwSTDP_userdef](#)
TODO This class definition may be code-generated in a future release.

Macros

- #define [DT](#) 0.1
This defines the global time step at which the simulation will run.

Functions

- void [modelDefinition](#) ([NNmodel](#) &model)
This function defines the MBody1 model with user defined synapses.

Variables

- int [nGPU](#) = 0
- float [myPOI_p](#) [4]
- float [myPOI_ini](#) [3]
- float [stdTM_p](#) [7]
- float [stdTM_ini](#) [4]
- float * [myPNKC_p](#) = NULL
- float [myPNKC_ini](#) [1]
- float [postExpPNKC](#) [2]
- float * [myPNLHI_p](#) = NULL
- float [myPNLHI_ini](#) [1]
- float [postExpPNLHI](#) [2]
- float [myLHIKC_p](#) [2]
- float [myLHIKC_ini](#) [1]
- float [postExpLHIKC](#) [2]
- float [myKCDN_p](#) [11]
- float [myKCDN_ini](#) [2]

- float `postExpKCDN` [2]
- float `myDNDN_p` [4]
- float `myDNDN_ini` [1]
- float `postExpDNDN` [2]
- float * `postSynV` = NULL
- float `postSynV_EXPDECAY_EVAR` [1]
- float * `gpPNKC` = new float[_NAL*_NMB]
- float * `gpKCDN` = new float[_NMB*_NLB]

11.56.1 Macro Definition Documentation

11.56.1.1 #define DT 0.1

This defines the global time step at which the simulation will run.

11.56.2 Function Documentation

11.56.2.1 void modelDefinition (NNmodel & model)

This function defines the MBody1 model with user defined synapses.

11.56.3 Variable Documentation

11.56.3.1 float* gpKCDN = new float[_NMB*_NLB]

11.56.3.2 float* gpPNKC = new float[_NAL*_NMB]

11.56.3.3 float myDNDN_ini[1]

Initial value:

```
= {
    5.0/_NLB
}
```

11.56.3.4 float myDNDN_p[4]

Initial value:

```
= {
    -30.0,
    50.0
}
```

11.56.3.5 float myKCDN_ini[2]

Initial value:

```
= {
    0.01,
    0.01,
}
```

11.56.3.6 float myKCDN_p[11]**Initial value:**

```
= {  
  -20.0,  
  50.0,  
  50.0,  
  50000.0,  
  100000.0,  
  200.0,  
  0.015,  
  0.0075,  
  33.33,  
  10.0,  
  0.00006  
}
```

11.56.3.7 float myLHIKC_ini[1]**Initial value:**

```
= {  
  0.35/_NLHI  
}
```

11.56.3.8 float myLHIKC_p[2]**Initial value:**

```
= {  
  -40.0,  
  50.0  
}
```

11.56.3.9 float myPNKC_ini[1]**Initial value:**

```
= {  
  0.01  
}
```

11.56.3.10 float* myPNKC_p = NULL**11.56.3.11 float myPNLHI_ini[1]****Initial value:**

```
= {  
  0.0  
}
```

11.56.3.12 float* myPNLHI_p = NULL**11.56.3.13 float myPOI_ini[3]****Initial value:**

```
= {  
  -60.0,  
  0,  
  -10.0  
}
```

11.56.3.14 float myPOI_p[4]

Initial value:

```
= {  
  0.1,  
  2.5,  
  20.0,  
  -60.0  
}
```

11.56.3.15 int nGPU = 0

11.56.3.16 float postExpDNDN[2]

Initial value:

```
= {  
  8.0,  
  -92.0  
}
```

11.56.3.17 float postExpKCDN[2]

Initial value:

```
= {  
  5.0,  
  0.0  
}
```

11.56.3.18 float postExpLHIKC[2]

Initial value:

```
= {  
  1.5,  
  -92.0  
}
```

11.56.3.19 float postExpPNKC[2]

Initial value:

```
= {  
  1.0,  
  0.0  
}
```

11.56.3.20 float postExpPNLHI[2]**Initial value:**

```
= {
  1.0,
  0.0
}
```

11.56.3.21 float* postSynV = NULL**11.56.3.22 float postSynV_EXPDECAY_EVAR[1]****Initial value:**

```
= {
  0
}
```

11.56.3.23 float stdTM_ini[4]**Initial value:**

```
= {
  -60.0,
  0.0529324,
  0.3176767,
  0.5961207
}
```

11.56.3.24 float stdTM_p[7]**Initial value:**

```
= {
  7.15,
  50.0,
  1.43,
  -95.0,
  0.02672,
  -63.563,
  0.143
}
```

11.57 tmp/test.cc File Reference

```
#include <cstdlib>
#include <iostream>
#include <stdint.h>
```

Macros

- #define `RAND`(Y, X) Y = Y * 1103515245 + 12345; X = (Y >> 16) & 281474976710655

Functions

- int `main` (void)

11.57.1 Macro Definition Documentation

11.57.1.1 `#define RAND(Y, X) Y = Y * 1103515245 + 12345; X = (Y >> 16) & 281474976710655`

11.57.2 Function Documentation

11.57.2.1 `int main (void)`

11.58 userproject/GeNNHelperKrnls.cu File Reference

```
#include <curand_kernel.h>
```

Macros

- `#define BlkSz 256`

Functions

- `__global__ void setup_kernel (curandState *state, unsigned long seed, int sizeofResult)`
- `__global__ void generate_random_gpulInput_xorwow (curandState *state, float *result, int sizeofResult, float Rstrength, float Rshift)`
- `void xorwow_setup (curandState *devStates, long int sampleSize)`

11.58.1 Macro Definition Documentation

11.58.1.1 `#define BlkSz 256`

11.58.2 Function Documentation

11.58.2.1 `__global__ void generate_random_gpulInput_xorwow (curandState * state, float * result, int sizeofResult, float Rstrength, float Rshift)`

11.58.2.2 `__global__ void setup_kernel (curandState * state, unsigned long seed, int sizeofResult)`

11.58.2.3 `void xorwow_setup (curandState * devStates, long int sampleSize)`

11.59 userproject/HHVclampGA_project/model/GA.cc File Reference

```
#include <algorithm>
```

Classes

- struct [errTupel](#)

Functions

- `int compareErrTupel (const void *x, const void *y)`
- `void procreatePop (FILE *osb)`

11.59.1 Function Documentation

11.59.1.1 `int compareErrTupel (const void * x, const void * y)`

11.59.1.2 `void procreatePop (FILE * osb)`

11.60 userproject/HHVclampGA_project/model/helper.h File Reference

```
#include <vector>
```

Classes

- struct [inputSpec](#)

Functions

- ostream & [operator<<](#) (ostream &os, [inputSpec](#) &l)
- void [write_para](#) ()
- void [single_var_reinit](#) (int n, double fac)
- void [copy_var](#) (int src, int trg)
- void [var_reinit](#) (double fac)
- void [truevar_init](#) ()
- void [initexpHH](#) ()
- void [truevar_initexpHH](#) ()
- void [runexpHH](#) (float t)
- void [initl](#) ([inputSpec](#) &l)

Variables

- double [sigGNa](#) = 0.1
- double [sigENa](#) = 10.0
- double [sigGK](#) = 0.1
- double [sigEK](#) = 10.0
- double [sigGl](#) = 0.1
- double [sigEl](#) = 10.0
- double [sigC](#) = 0.1
- double [Vexp](#)
- double [mexp](#)
- double [hexp](#)
- double [nexp](#)
- double [gNaexp](#)
- double [ENaexp](#)
- double [gKexp](#)
- double [EKexp](#)
- double [glexp](#)
- double [Elexp](#)
- double [Cexp](#)

11.60.1 Function Documentation

11.60.1.1 void copy_var (int *src*, int *trg*)

11.60.1.2 void initexpHH ()

11.60.1.3 void initl (inputSpec & *l*)

11.60.1.4 ostream& operator<< (ostream & *os*, inputSpec & *l*)

11.60.1.5 void runexpHH (float *t*)

11.60.1.6 void single_var_reinit (int *n*, double *fac*)

11.60.1.7 void truevar_init ()

11.60.1.8 void truevar_initexpHH ()

11.60.1.9 void var_reinit (double *fac*)

11.60.1.10 void write_para ()

11.60.2 Variable Documentation

11.60.2.1 double Cexp

11.60.2.2 double EKexp

11.60.2.3 double Elexp

11.60.2.4 double ENaexp

11.60.2.5 double gKexp

11.60.2.6 double glexp

11.60.2.7 double gNaexp

11.60.2.8 double hexp

11.60.2.9 double mexp

11.60.2.10 double nexp

11.60.2.11 double sigC = 0.1

11.60.2.12 double sigEK = 10.0

11.60.2.13 double sigEl = 10.0

11.60.2.14 double sigENa = 10.0

11.60.2.15 double sigGK = 0.1

11.60.2.16 double sigGl = 0.1

11.60.2.17 double sigGNa = 0.1

11.60.2.18 double Vexp

11.61 userproject/HHVclampGA_project/model/HHVClamp.cc File Reference

This file contains the model definition of HHVClamp model. It is used in both the GeNN code generation and the user side simulation code. The HHVClamp model implements a population of unconnected Hodgkin-Huxley neurons that evolve to mimick a model run on the CPU, using genetic algorithm techniques.

```
#include "modelSpec.h"
#include "modelSpec.cc"
#include "HHVClampParameters.h"
```

Macros

- `#define DT 0.5`

This defines the global time step at which the simulation will run.

Functions

- void `modelDefinition (NNmodel &model)`

This function defines the HH model with variable parameters.

Variables

- float `myHH_ini` [11]
- float * `myHH_p` = NULL

11.61.1 Detailed Description

This file contains the model definition of HHVClamp model. It is used in both the GeNN code generation and the user side simulation code. The HHVClamp model implements a population of unconnected Hodgkin-Huxley neurons that evolve to mimick a model run on the CPU, using genetic algorithm techniques.

11.61.2 Macro Definition Documentation

11.61.2.1 `#define DT 0.5`

This defines the global time step at which the simulation will run.

11.61.3 Function Documentation

11.61.3.1 void `modelDefinition (NNmodel & model)`

This function defines the HH model with variable parameters.

11.61.4 Variable Documentation

11.61.4.1 float myHH_ini[11]

Initial value:

```
= {
  -60.0,
  0.0529324,
  0.3176767,
  0.5961207,
  120.0,
  55.0,
  36.0,
  -72.0,
  0.3,
  -50.0,
  1.0
}
```

11.61.4.2 float* myHH_p = NULL

11.62 userproject/HHVclampGA_project/model/MBody1.cc File Reference

```
#include "modelSpec.h"
#include "modelSpec.cc"
#include "../userproject/include/sizes.h"
```

Macros

- `#define DT 0.1`
This defines the global time step at which the simulation will run.

Functions

- void `modelDefinition (NNmodel &model)`
This function defines the MBody1 model, and it is a good example of how networks should be defined.

Variables

- float `myPOI_p` [4]
- float `myPOI_ini` [4]
- float `stdTM_p` [7]
- float `stdTM_ini` [4]
- float `myPNKC_p` [3]
- float `postExpPNKC` [2]
- float `myPNLHI_p` [3]
- float `postExpPNLHI` [2]
- float `myLHIKC_p` [4]
- float `gLHIKC` = 0.006
- float `postExpLHIKC` [2]
- float `myKCDN_p` [13]
- float `postExpKCDN` [2]
- float `myDNDN_p` [4]
- float `gDNDN` = 0.01
- float `postExpDNDN` [2]
- float * `postSynV` = NULL

11.62.1 Macro Definition Documentation

11.62.1.1 #define DT 0.1

This defines the global time step at which the simulation will run.

11.62.2 Function Documentation

11.62.2.1 void modelDefinition (NNmodel & *model*)

This function defines the MBody1 model, and it is a good example of how networks should be defined.

11.62.3 Variable Documentation

11.62.3.1 float gDNDN = 0.01

11.62.3.2 float gLHIKC = 0.006

11.62.3.3 float myDNDN_p[4]

Initial value:

```
= {
  -92.0,
  -30.0,
   8.0,
  50.0
}
```

11.62.3.4 float myKCDN_p[13]

Initial value:

```
= {
  0.0,
 -20.0,
  5.0,
 25.0,
100.0,
50000.0,
100000.0,
100.0,
 0.06,
 0.03,
33.33,
10.0,
0.00006
}
```

11.62.3.5 float myLHIKC_p[4]

Initial value:

```
= {
  -92.0,
 -40.0,
  3.0,
 50.0
}
```

11.62.3.6 float myPNKC_p[3]

Initial value:

```
= {  
    0.0,  
    -20.0,  
    1.0  
}
```

11.62.3.7 float myPNLHI_p[3]

Initial value:

```
= {  
    0.0,  
    -20.0,  
    1.0  
}
```

11.62.3.8 float myPOI_ini[4]

Initial value:

```
= {  
    -60.0,  
    0,  
    -10.0,  
}
```

11.62.3.9 float myPOI_p[4]

Initial value:

```
= {  
    0.1,  
    2.5,  
    20.0,  
    -60.0  
}
```

11.62.3.10 float postExpDNDN[2]

Initial value:

```
= {  
    8.0,  
    -92.0  
}
```

11.62.3.11 float postExpKCDN[2]

Initial value:

```
= {  
    5.0,  
    0.0  
}
```

11.62.3.12 float postExpLHIKC[2]

Initial value:

```
= {  
    3.0,  
    -92.0  
}
```

11.62.3.13 float postExpPNKC[2]

Initial value:

```
= {  
    1.0,  
    0.0  
}
```

11.62.3.14 float postExpPNLHI[2]

Initial value:

```
= {  
    1.0,  
    0.0  
}
```

11.62.3.15 float* postSynV = NULL

11.62.3.16 float stdTM_ini[4]

Initial value:

```
= {  
    -60.0,  
    0.0529324,  
    0.3176767,  
    0.5961207  
}
```

11.62.3.17 float stdTM_p[7]

Initial value:

```
= {  
    7.15,  
    50.0,  
    1.43,  
    -95.0,  
    0.02672,  
    -63.563,  
    0.143  
}
```

11.63 userproject/MBody1_project/model/MBody1.cc File Reference

```
#include "modelSpec.h"  
#include "modelSpec.cc"  
#include "sizes.h"
```

Macros

- `#define DT 0.1`

This defines the global time step at which the simulation will run.

Functions

- `void modelDefinition (NNmodel &model)`

This function defines the MBody1 model, and it is a good example of how networks should be defined.

Variables

- `int nGPU = 0`
- `float myPOI_p [4]`
- `float myPOI_ini [3]`
- `float stdTM_p [7]`
- `float stdTM_ini [4]`
- `float * myPNKC_p = NULL`
- `float myPNKC_ini [1]`
- `float postExpPNKC [2]`
- `float * myPNLHI_p = NULL`
- `float myPNLHI_ini [1]`
- `float postExpPNLHI [2]`
- `float myLHIKC_p [2]`
- `float myLHIKC_ini [1]`
- `float postExpLHIKC [2]`
- `float myKCDN_p [11]`
- `float myKCDN_ini [2]`
- `float postExpKCDN [2]`
- `float myDNDN_p [2]`
- `float myDNDN_ini [1]`
- `float postExpDNDN [2]`
- `float * postSynV = NULL`

11.63.1 Macro Definition Documentation

11.63.1.1 `#define DT 0.1`

This defines the global time step at which the simulation will run.

11.63.2 Function Documentation

11.63.2.1 `void modelDefinition (NNmodel & model)`

This function defines the MBody1 model, and it is a good example of how networks should be defined.

11.63.3 Variable Documentation

11.63.3.1 float myDNDN_ini[1]

Initial value:

```
= {  
    5.0/_NLB  
}
```

11.63.3.2 float myDNDN_p[2]

Initial value:

```
= {  
    -30.0,  
    50.0  
}
```

11.63.3.3 float myKCDN_ini[2]

Initial value:

```
= {  
    0.01,  
    0.01,  
}
```

11.63.3.4 float myKCDN_p[11]

Initial value:

```
= {  
    -20.0,  
    50.0,  
    50.0,  
    50000.0,  
    100000.0,  
    200.0,  
    0.015,  
    0.0075,  
    33.33,  
    10.0,  
    0.00006  
}
```

11.63.3.5 float myLHIKC_ini[1]

Initial value:

```
= {  
    0.35/_NLHI  
}
```

11.63.3.6 float myLHIKC_p[2]

Initial value:

```
= {  
    -40.0,  
    50.0  
}
```


11.63.3.7 float myPNKC_ini[1]

Initial value:

```
= {  
    0.01  
}
```

11.63.3.8 float* myPNKC_p = NULL

11.63.3.9 float myPNLHI_ini[1]

Initial value:

```
= {  
    0.0  
}
```

11.63.3.10 float* myPNLHI_p = NULL

11.63.3.11 float myPOI_ini[3]

Initial value:

```
= {  
    -60.0,  
    0,  
    -10.0  
}
```

11.63.3.12 float myPOI_p[4]

Initial value:

```
= {  
    0.1,  
    2.5,  
    20.0,  
    -60.0  
}
```

11.63.3.13 int nGPU = 0

11.63.3.14 float postExpDNDN[2]

Initial value:

```
= {  
    8.0,  
    -92.0  
}
```

11.63.3.15 float postExpKCDN[2]

Initial value:

```
= {  
    5.0,  
    0.0  
}
```

11.63.3.16 float postExpLHIKC[2]

Initial value:

```
= {  
    1.5,  
   -92.0  
}
```

11.63.3.17 float postExpPNKC[2]

Initial value:

```
= {  
    1.0,  
    0.0  
}
```

11.63.3.18 float postExpPNLHI[2]

Initial value:

```
= {  
    1.0,  
    0.0  
}
```

11.63.3.19 float* postSynV = NULL

11.63.3.20 float stdTM_ini[4]

Initial value:

```
= {  
   -60.0,  
   0.0529324,  
   0.3176767,  
   0.5961207  
}
```

11.63.3.21 float stdTM_p[7]

Initial value:

```
= {  
    7.15,  
   50.0,  
    1.43,  
   -95.0,  
    0.02672,  
   -63.563,  
    0.143  
}
```

11.64 userproject/MBody1_project/modelcmp/MBody1.cc File Reference

```
#include "modelSpec.h"  
#include "modelSpec.cc"  
#include "../userproject/include/sizes.h"
```

Macros

- `#define DT 0.1`

This defines the global time step at which the simulation will run.

Functions

- void `modelDefinition` (`NNmodel` &`model`)

This function defines the MBody1 model, and it is a good example of how networks should be defined.

Variables

- int `nGPU` = 0
- float `myPOI_p` [4]
- float `myPOI_ini` [4]
- float `stdTM_p` [7]
- float `stdTM_ini` [4]
- float `myPNKC_p` [3]
- float `postExpPNKC` [2]
- float `myPNLHI_p` [3]
- float `postExpPNLHI` [2]
- float `myLHIKC_p` [4]
- float `gLHIKC` = 0.35/_NLHI
- float `postExpLHIKC` [2]
- float `myKCDN_p` [13]
- float `postExpKCDN` [2]
- float `myDNDN_p` [4]
- float `gDNDN` = 1.0/_NLB
- float `postExpDNDN` [2]
- float * `postSynV` = NULL

11.64.1 Macro Definition Documentation

11.64.1.1 `#define DT 0.1`

This defines the global time step at which the simulation will run.

11.64.2 Function Documentation

11.64.2.1 void `modelDefinition` (`NNmodel` & `model`)

This function defines the MBody1 model, and it is a good example of how networks should be defined.

11.64.3 Variable Documentation

11.64.3.1 float `gDNDN` = 1.0/_NLB

11.64.3.2 float `gLHIKC` = 0.35/_NLHI

11.64.3.3 float `myDNDN_p`[4]

Initial value:

```
= {  
  -92.0,  
  -30.0,  
  8.0,  
  50.0  
}
```

11.64.3.4 float myKCDN_p[13]

Initial value:

```
= {  
  0.0,  
  -20.0,  
  5.0,  
  50.0,  
  50.0,  
  50000.0,  
  100000.0,  
  200.0,  
  0.015,  
  0.0075,  
  33.33,  
  10.0,  
  
  0.00006  
}
```

11.64.3.5 float myLHIKC_p[4]

Initial value:

```
= {  
  -92.0,  
  -40.0,  
  1.5,  
  50.0  
}
```

11.64.3.6 float myPNKC_p[3]

Initial value:

```
= {  
  5.0,  
  -20.0,  
  1.0  
}
```

11.64.3.7 float myPNLHI_p[3]

Initial value:

```
= {  
  0.0,  
  -20.0,  
  1.0  
}
```

11.64.3.8 float myPOI_ini[4]

Initial value:

```
= {  
  -60.0,  
  0,  
  -10.0,  
}
```

11.64.3.9 float myPOI_p[4]

Initial value:

```
= {  
  0.1,  
  2.5,  
  20.0,  
  -60.0  
}
```

11.64.3.10 int nGPU = 0

11.64.3.11 float postExpDNDN[2]

Initial value:

```
= {  
  8.0,  
  -92.0  
}
```

11.64.3.12 float postExpKCDN[2]

Initial value:

```
= {  
  5.0,  
  0.0  
}
```

11.64.3.13 float postExpLHIKC[2]

Initial value:

```
= {  
  1.5,  
  -92.0  
}
```

11.64.3.14 float postExpPNKC[2]

Initial value:

```
= {  
  1.0,  
  0.0  
}
```

11.64.3.15 float postExpPNLHI[2]**Initial value:**

```
= {
    1.0,
    0.0
}
```

11.64.3.16 float* postSynV = NULL**11.64.3.17 float stdTM_ini[4]****Initial value:**

```
= {
    -60.0,
    0.0529324,
    0.3176767,
    0.5961207
}
```

11.64.3.18 float stdTM_p[7]**Initial value:**

```
= {
    7.15,
    50.0,
    1.43,
    -95.0,
    0.02672,
    -63.563,
    0.143
}
```

11.65 userproject/HHVclampGA_project/model/VClampGA.cu File Reference

Main entry point for the GeNN project demonstrating realtime fitting of a neuron with a GA running mostly on the GPU.

```
#include "VClampGA.h"
```

Functions

- `int main (int argc, char *argv[])`
This function is the entry point for running the project.

11.65.1 Detailed Description

Main entry point for the GeNN project demonstrating realtime fitting of a neuron with a GA running mostly on the GPU.

11.65.2 Function Documentation**11.65.2.1 int main (int argc, char * argv[])**

This function is the entry point for running the project.

11.66 userproject/HHVclampGA_project/model/VClampGA.h File Reference

```
#include <cassert>
#include "hr_time.cpp"
#include "utils.h"
#include <cuda_runtime.h>
#include "HHVClamp.cc"
#include "HHVClamp_CODE/runner.cc"
#include "../.../lib/include/numlib/randomGen.h"
#include "../.../lib/include/numlib/gauss.h"
#include "helper.h"
#include "GA.cc"
```

Macros

- `#define RAND(Y, X) Y = Y * 1103515245 + 12345; X = (unsigned int)(Y >> 16) & 32767`

Variables

- `randomGen R`
- `randomGauss RG`
- `double t = 0.0f`
- `unsigned int iT = 0`
- `CStopWatch timer`

11.66.1 Macro Definition Documentation

11.66.1.1 `#define RAND(Y, X) Y = Y * 1103515245 + 12345; X = (unsigned int)(Y >> 16) & 32767`

11.66.2 Variable Documentation

11.66.2.1 `unsigned int iT = 0`

11.66.2.2 `randomGen R`

11.66.2.3 `randomGauss RG`

11.66.2.4 `double t = 0.0f`

11.66.2.5 `CStopWatch timer`

11.67 userproject/include/HHVClampParameters.h File Reference

Macros

- `#define NPOP 5000`
- `#define TOTALT 1000`

11.67.1 Macro Definition Documentation

11.67.1.1 `#define NPOP 5000`

11.67.1.2 `#define TOTALT 1000`

11.68 userproject/include/sizes.h File Reference

Macros

- `#define _NAL 100`
- `#define _NMB 1000`
- `#define _NLHI 20`
- `#define _NLB 100`
- `#define _FTYPE FLOAT`
- `#define scalar float`
- `#define SCALAR_MIN FLT_MIN`
- `#define SCALAR_MAX FLT_MAX`

11.68.1 Macro Definition Documentation

11.68.1.1 `#define _FTYPE FLOAT`

11.68.1.2 `#define _NAL 100`

11.68.1.3 `#define _NLB 100`

11.68.1.4 `#define _NLHI 20`

11.68.1.5 `#define _NMB 1000`

11.68.1.6 `#define scalar float`

11.68.1.7 `#define SCALAR_MAX FLT_MAX`

11.68.1.8 `#define SCALAR_MIN FLT_MIN`

11.69 userproject/lzh_sparse_project/model/lzh_sim_sparse.cu File Reference

```
#include <iostream>
#include <fstream>
#include "Izh_sparse_sim.h"
#include "../GeNNHelperKrnls.cu"
```

Functions

- `int main (int argc, char *argv[])`

11.69.1 Function Documentation

11.69.1.1 `int main (int argc, char * argv[])`

11.70 userproject/lzh_sparse_project/model/lzh_sparse.cc File Reference

```
#include "modelSpec.h"
#include "modelSpec.cc"
#include <vector>
#include "../..//userproject/include/sizes.h"
```

Macros

- `#define DT 1.0`

Functions

- void `modelDefinition` (`NNmodel` &`model`)

Variables

- `std::vector< unsigned int >` `neuronPSize`
- `std::vector< unsigned int >` `neuronVSize`
- `std::vector< unsigned int >` `synapsePSize`
- `float *` `exclzh_p` = NULL
- `float *` `inhlzh_p` = NULL
- `float` `lzhExc_ini` [6]
- `float` `lzhInh_ini` [6]
- `float *` `Synlzh_p` = NULL
- `float` `postExpP` [2]
- `float *` `postSynV` = NULL
- `float` `Synlzh_ini` [1]

11.70.1 Macro Definition Documentation

11.70.1.1 `#define DT 1.0`

11.70.2 Function Documentation

11.70.2.1 void `modelDefinition` (`NNmodel` & *model*)

11.70.3 Variable Documentation

11.70.3.1 `float*` `exclzh_p` = NULL

11.70.3.2 `float*` `inhlzh_p` = NULL

11.70.3.3 `float` `lzhExc_ini`[6]

Initial value:

```
= {
    -65.0,
    0.0,
    0.02,
    0.2,
    -65.0,
    8.0
}
```

11.70.3.4 float IzhInh_ini[6]**Initial value:**

```
= {
    -65,
    0.0,
    0.02,
    0.25,
    -65.0,
    2.0
}
```

11.70.3.5 std::vector<unsigned int> neuronPSize**11.70.3.6 std::vector<unsigned int> neuronVSize****11.70.3.7 float postExpP[2]****Initial value:**

```
= {
    0.0,
    0.0
}
```

11.70.3.8 float* postSynV = NULL**11.70.3.9 std::vector<unsigned int> synapsePSize****11.70.3.10 float SynIzh_ini[1]****Initial value:**

```
= {
    0.0
}
```

11.70.3.11 float* SynIzh_p = NULL**11.71 userproject/Izh_sparse_project/model/Izh_sparse_model.cc File Reference**

```
#include "Izh_sparse_CODE/runner.cc"
#include "../lib/include/numlib/randomGen.h"
#include "../lib/include/numlib/gauss.h"
```

Macros

- `#define IZH_SPARSE_MODEL_CC_`

Variables

- `randomGauss RG`
- `randomGen R`

11.71.1 Macro Definition Documentation

11.71.1.1 `#define _IZH_SPARSE_MODEL_CC_`

11.71.2 Variable Documentation

11.71.2.1 `randomGen R`

11.71.2.2 `randomGauss RG`

11.72 userproject/lzh_sparse_project/model/lzh_sparse_model.h File Reference

```
#include "Izh_sparse.cc"
```

Classes

- class [classIzh](#)

11.73 userproject/lzh_sparse_project/model/lzh_sparse_sim.h File Reference

```
#include <cassert>
#include "hr_time.cpp"
#include "utils.h"
#include <cuda_runtime.h>
#include "Izh_sparse_model.h"
#include "Izh_sparse_model.cc"
```

Macros

- `#define DBG_SIZE 5000`
- `#define T_REPORT_TME 5000.0`
- `#define TOTAL_TME 5000.0`

Variables

- float `t` = 0.0f
- unsigned int `iT` = 0
- [CStopWatch](#) `timer`

11.73.1 Macro Definition Documentation

11.73.1.1 `#define DBG_SIZE 5000`

11.73.1.2 `#define T_REPORT_TME 5000.0`

11.73.1.3 `#define TOTAL_TME 5000.0`

11.73.2 Variable Documentation

11.73.2.1 unsigned int iT = 0

11.73.2.2 float t = 0.0f

11.73.2.3 CStopWatch timer

11.74 userproject/MBody1_project/model/classol_sim_detailed_output.cu File Reference

```
#include "classol_sim.h"
```

Classes

- struct [histogram](#)

Functions

- bool [compare](#) ([histogram](#) a, [histogram](#) b)
- int [main](#) (int argc, char *argv[])

This function is the entry point for running the simulation of the MBody1 model network.

11.74.1 Function Documentation

11.74.1.1 bool [compare](#) ([histogram](#) a, [histogram](#) b)

11.74.1.2 int [main](#) (int *argc*, char * *argv*[])

This function is the entry point for running the simulation of the MBody1 model network.

11.75 userproject/OneComp_project/model/OneComp.cc File Reference

```
#include "modelSpec.h"
#include "modelSpec.cc"
#include "../userproject/include/sizes.h"
```

Macros

- #define [DT](#) 1.0

Functions

- void [modelDefinition](#) ([NNmodel](#) &model)

Variables

- float [exlzh_p](#) [4]
- float [exlzh_ini](#) [2]
- float [mySyn_p](#) [3]
- float [postExp](#) [2]

- float * `postSynV` = NULL
- float `inplzh1` = 4.0

11.75.1 Macro Definition Documentation

11.75.1.1 `#define DT 1.0`

11.75.2 Function Documentation

11.75.2.1 `void modelDefinition (NNmodel & model)`

11.75.3 Variable Documentation

11.75.3.1 `float exlzh_ini[2]`

Initial value:

```
= {
    -65,
    -20
}
```

11.75.3.2 `float exlzh_p[4]`

Initial value:

```
= {
    0.02,
    0.2,
    -65,
    6
}
```

11.75.3.3 `float inplzh1 = 4.0`

11.75.3.4 `float mySyn_p[3]`

Initial value:

```
= {
    0.0,
    -20.0,
    1.0
}
```

11.75.3.5 `float postExp[2]`

Initial value:

```
= {
    1.0,
    0.0
}
```

11.75.3.6 `float* postSynV = NULL`

11.76 `userproject/OneComp_project/model/OneComp_model.cc` File Reference

```
#include "OneComp_CODE/runner.cc"
```

Macros

- `#define _ONECOMP_MODEL_CC_`

11.76.1 Macro Definition Documentation

11.76.1.1 `#define _ONECOMP_MODEL_CC_`

11.77 `userproject/OneComp_project/model/OneComp_model.h` File Reference

```
#include "OneComp.cc"
```

Classes

- class [classol](#)

This class cponains the methods for running the MBody1 example model.

11.78 `userproject/OneComp_project/model/OneComp_sim.cu` File Reference

```
#include "OneComp_sim.h"
```

Functions

- int [main](#) (int argc, char *argv[])

11.78.1 Function Documentation

11.78.1.1 `int main (int argc, char * argv[])`

11.79 `userproject/OneComp_project/model/OneComp_sim.h` File Reference

```
#include <cassert>
#include "hr_time.cpp"
#include "utils.h"
#include <cuda_runtime.h>
#include "OneComp_model.h"
#include "OneComp_model.cc"
```

Macros

- `#define DBG_SIZE 10000`
- `#define T_REPORT_TME 100.0`
- `#define TOTAL_TME 5000`

Variables

- float `t` = 0.0f
- unsigned int `iT` = 0
- `CStopWatch` timer

11.79.1 Macro Definition Documentation

11.79.1.1 `#define DBG_SIZE 10000`

11.79.1.2 `#define T_REPORT_TME 100.0`

11.79.1.3 `#define TOTAL_TME 5000`

11.79.2 Variable Documentation

11.79.2.1 unsigned int `iT` = 0

11.79.2.2 float `t` = 0.0f

11.79.2.3 `CStopWatch` timer

11.80 userproject/PoissonIzh_project/model/PoissonIzh-model.cc File Reference

```
#include "PoissonIzh_CODE/runner.cc"
```

Macros

- `#define _POISSONIZHMODEL_CC_`

11.80.1 Macro Definition Documentation

11.80.1.1 `#define _POISSONIZHMODEL_CC_`

11.81 userproject/PoissonIzh_project/model/PoissonIzh-model.h File Reference

```
#include "PoissonIzh.cc"
```

Classes

- class `classol`

This class contains the methods for running the MBody1 example model.

11.82 userproject/PoissonIzh_project/model/PoissonIzh.cc File Reference

```
#include "modelSpec.h"
#include "modelSpec.cc"
#include "../..//userproject/include/sizes.h"
```

Macros

- `#define DT 1.0`

Functions

- `void modelDefinition (NNmodel &model)`

Variables

- `float myPOI_p [4]`
- `float myPOI_ini [4]`
- `float exlzh_p [4]`
- `float exlzh_ini [2]`
- `float mySyn_p [3]`
- `float postExp [2]`
- `float * postSynV = NULL`

11.82.1 Macro Definition Documentation

11.82.1.1 `#define DT 1.0`

11.82.2 Function Documentation

11.82.2.1 `void modelDefinition (NNmodel & model)`

11.82.3 Variable Documentation

11.82.3.1 `float exlzh_ini[2]`

Initial value:

```
= {
    -65,
    -20
}
```

11.82.3.2 `float exlzh_p[4]`

Initial value:

```
= {
    0.02,
    0.2,
    -65,
    6
}
```


11.82.3.3 float myPOI_ini[4]

Initial value:

```
= {  
    -60.0,  
    0,  
    -10.0  
}
```

11.82.3.4 float myPOI_p[4]

Initial value:

```
= {  
    1,  
    2.5,  
    20.0,  
    -60.0  
}
```

11.82.3.5 float mySyn_p[3]

Initial value:

```
= {  
    0.0,  
    -20.0,  
    1.0  
}
```

11.82.3.6 float postExp[2]

Initial value:

```
= {  
    1.0,  
    0.0  
}
```

11.82.3.7 float* postSynV = NULL

11.83 userproject/PoissonIzh_project/model/PoissonIzh_sim.cu File Reference

```
#include "PoissonIzh_sim.h"
```

Functions

- int [main](#) (int argc, char *argv[])

11.83.1 Function Documentation

11.83.1.1 `int main (int argc, char * argv[])`

11.84 userproject/PoissonIzh_project/model/PoissonIzh_sim.h File Reference

```
#include <cassert>
#include "hr_time.cpp"
#include "utils.h"
#include <cuda_runtime.h>
#include "PoissonIzh-model.h"
#include "PoissonIzh-model.cc"
```

Macros

- `#define RAND(Y, X) Y = Y * 1103515245 + 12345; X = (unsigned int)(Y >> 16) & 32767`
- `#define INJECTCURRENT 0`
- `#define DBG_SIZE 1000`
- `#define INJECTCURRENT 0`
- `#define PATTERNNO 100`
- `#define INPUTBASERATE 17`
- `#define T_REPORT_TME 1000.0`
- `#define SYN_OUT_TME 2000.0`
- `#define PAT_TIME 100.0`
- `#define PATFTIME 1.0`
- `#define TOTAL_TME 5000`

Variables

- float `t` = 0.0f
- unsigned int `iT` = 0
- int `patSetTime`
- int `patFireTime`
- `CStopWatch` timer

11.84.1 Macro Definition Documentation

11.84.1.1 `#define DBG_SIZE 1000`

11.84.1.2 `#define INJECTCURRENT 0`

11.84.1.3 `#define INJECTCURRENT 0`

11.84.1.4 `#define INPUTBASERATE 17`

11.84.1.5 `#define PAT_TIME 100.0`

11.84.1.6 `#define PATFTIME 1.0`

11.84.1.7 `#define PATTERNNO 100`

11.84.1.8 `#define RAND(Y, X) Y = Y * 1103515245 + 12345; X = (unsigned int)(Y >> 16) & 32767`

11.84.1.9 `#define SYN_OUT_TME 2000.0`

11.84.1.10 `#define T_REPORT_TME 1000.0`

11.84.1.11 `#define TOTAL_TME 5000`

11.84.2 Variable Documentation

11.84.2.1 `unsigned int iT = 0`

11.84.2.2 `int patFireTime`

11.84.2.3 `int patSetTime`

11.84.2.4 `float t = 0.0f`

11.84.2.5 `CStopWatch` timer

11.85 userproject/SynDelay_project/SynDelay.cc File Reference

```
#include "modelSpec.h"  
#include "modelSpec.cc"
```

Macros

- `#define DT 1.0f`

Functions

- `void modelDefinition (NNmodel &model)`

Variables

- `float input_p [4]`
- `float input_ini [2]`
- `float constInput = 4.0`
- `float postExplnp [2]`
- `float inter_p [4]`
- `float inter_ini [2]`
- `float postExplnt [2]`
- `float output_p [4]`
- `float output_ini [2]`
- `float postExpOut [2]`
- `float synapses_p [3]`
- `float * postSynV = NULL`
- `float strongSynG = 0.0006`
- `float weakSynG = 0.0002`

11.85.1 Macro Definition Documentation

11.85.1.1 #define DT 1.0f

11.85.2 Function Documentation

11.85.2.1 void modelDefinition (NNmodel & *model*)

11.85.3 Variable Documentation

11.85.3.1 float constInput = 4.0

11.85.3.2 float input_ini[2]

Initial value:

```
= {
    -65,
    -20
}
```

11.85.3.3 float input_p[4]

Initial value:

```
= {
    0.02,
    0.2,
    -65,
    6
}
```

11.85.3.4 float inter_ini[2]

Initial value:

```
= {
    -65,
    -20
}
```

11.85.3.5 float inter_p[4]

Initial value:

```
= {
    0.02,
    0.2,
    -65,
    6
}
```

11.85.3.6 float output_ini[2]

Initial value:

```
= {
    -65,
    -20
}
```

11.85.3.7 float output_p[4]

Initial value:

```
= {  
    0.02,  
    0.2,  
    -65,  
    6  
}
```

11.85.3.8 float postExpInp[2]

Initial value:

```
= {  
    1.0,  
    0.0  
}
```

11.85.3.9 float postExpInt[2]

Initial value:

```
= {  
    1.0,  
    0.0  
}
```

11.85.3.10 float postExpOut[2]

Initial value:

```
= {  
    1.0,  
    0.0  
}
```

11.85.3.11 float* postSynV = NULL

11.85.3.12 float strongSynG = 0.0006

11.85.3.13 float synapses_p[3]

Initial value:

```
= {  
    0.0,  
    -30.0,  
    1.0  
}
```

11.85.3.14 float weakSynG = 0.0002

11.86 userproject/SynDelay_project/SynDelaySim.cu File Reference

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include "hr_time.cpp"
#include "utils.h"
#include "SynDelaySim.h"
#include "SynDelay_CODE/runner.cc"
```

Macros

- `#define` [SYNDELAYSIM_CU](#)

Functions

- int [main](#) (int argc, char *argv[])

11.86.1 Macro Definition Documentation

11.86.1.1 `#define` [SYNDELAYSIM_CU](#)

11.86.2 Function Documentation

11.86.2.1 int main (int *argc*, char * *argv*[])

11.87 userproject/SynDelay_project/SynDelaySim.h File Reference

Classes

- class [SynDelay](#)

Macros

- `#define` [DT](#) 1.0f
- `#define` [TOTAL_TIME](#) 5000.0f
- `#define` [REPORT_TIME](#) 1000.0f

11.87.1 Macro Definition Documentation

11.87.1.1 `#define` [DT](#) 1.0f

11.87.1.2 `#define` [REPORT_TIME](#) 1000.0f

11.87.1.3 `#define` [TOTAL_TIME](#) 5000.0f

11.88 userproject/tools/gen_input_structured.cc File Reference

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include "randomGen.h"
#include "randomGen.cc"
```

Functions

- int [main](#) (int argc, char *argv[])

Variables

- [randomGen R](#)

11.88.1 Function Documentation

11.88.1.1 int main (int *argc*, char * *argv*[])

11.88.2 Variable Documentation

11.88.2.1 [randomGen R](#)

11.89 userproject/tools/gen_kcdn_syns.cc File Reference

This file is part of a tool chain for running the classol/MBody1 example model.

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include "randomGen.h"
#include "gauss.h"
#include "randomGen.cc"
```

Functions

- int [main](#) (int argc, char *argv[])

Variables

- [randomGen R](#)
- [randomGauss RG](#)

11.89.1 Detailed Description

This file is part of a tool chain for running the classol/MBody1 example model.

This file compiles to a tool to generate appropriate connectivity patterns between KCs and DN_s (detector neurons) in the model. The connectivity is saved to file and can then be read by the classol method for reading this connectivity.

11.89.2 Function Documentation

11.89.2.1 `int main (int argc, char * argv[])`

11.89.3 Variable Documentation

11.89.3.1 `randomGen R`

11.89.3.2 `randomGauss RG`

11.90 userproject/tools/gen_pnkc_syns.cc File Reference

This file is part of a tool chain for running the classol/MBody1 example model.

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include "randomGen.h"
#include "gauss.h"
#include "randomGen.cc"
```

Functions

- `int main (int argc, char *argv[])`

Variables

- `randomGen R`
- `randomGauss RG`

11.90.1 Detailed Description

This file is part of a tool chain for running the classol/MBody1 example model.

This file compiles to a tool to generate appropriate connectivity patterns between PNs and KCs in the model. The connectivity is saved to file and can then be read by the classol method for reading this connectivity.

11.90.2 Function Documentation

11.90.2.1 `int main (int argc, char * argv[])`

11.90.3 Variable Documentation

11.90.3.1 `randomGen R`

11.90.3.2 `randomGauss RG`

11.91 userproject/tools/gen_pnkc_syns_indivID.cc File Reference

This file is part of a tool chain for running the classol/MBody1 example model.


```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include "randomGen.h"
#include "gauss.h"
#include "simpleBit.h"
#include "randomGen.cc"
```

Functions

- `int main (int argc, char *argv[])`

Variables

- `randomGen R`
- `randomGauss RG`

11.91.1 Detailed Description

This file is part of a tool chain for running the classol/MBody1 example model.

This file compiles to a tool to generate appropriate connectivity patterns between PNs and KCs in the model. In contrast to the `gen_pnkc_syns.cc` tool, here the output is in a format that is suited for the "INDIVIDUALID" method for specifying connectivity. The connectivity is saved to file and can then be read by the classol method for reading this connectivity.

11.91.2 Function Documentation

11.91.2.1 `int main (int argc, char * argv[])`

11.91.3 Variable Documentation

11.91.3.1 `randomGen R`

11.91.3.2 `randomGauss RG`

11.92 userproject/tools/gen_pnlhi_syns.cc File Reference

This file is part of a tool chain for running the classol/MBody1 example model.

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
```

Functions

- `int main (int argc, char *argv[])`

11.92.1 Detailed Description

This file is part of a tool chain for running the classol/MBody1 example model.

This file compiles to a tool to generate appropriate connectivity patterns between PNs and LHIs (lateral horn interneurons) in the model. The connectivity is saved to file and can then be read by the classol method for reading this connectivity.

11.92.2 Function Documentation

11.92.2.1 `int main (int argc, char * argv[])`

11.93 userproject/tools/gen_syms_sparse.cc File Reference

This file generates the arrays needed for sparse connectivity. The connectivity is saved to a file for each variable and can then be read to fill the struct of connectivity.

```
#include <iostream>
#include <fstream>
#include <string.h>
#include "randomGen.h"
#include "gauss.h"
#include <vector>
```

Functions

- `int main` (int *argc*, char **argv*[])

Variables

- `randomGen R`
- `randomGauss RG`

11.93.1 Detailed Description

This file generates the arrays needed for sparse connectivity. The connectivity is saved to a file for each variable and can then be read to fill the struct of connectivity.

11.93.2 Function Documentation

11.93.2.1 `int main (int argc, char * argv[])`

11.93.3 Variable Documentation

11.93.3.1 `randomGen R`

11.93.3.2 `randomGauss RG`

11.94 userproject/tools/gen_syms_sparse_izhModel.cc File Reference

This file is part of a tool chain for running the Izhikevich network model.

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <string.h>
#include <vector>
#include "randomGen.h"
#include "randomGen.cc"
```

Functions

- int [printVector](#) (vector< unsigned int > &)
- int [printVector](#) (vector< double > &)
- int [main](#) (int argc, char *argv[])

Variables

- [randomGen R](#)
- [randomGen Rind](#)
- double [gsyn](#)
- double * [garray](#)
- unsigned int * [ind](#)
- double * [garray_ee](#)
- std::vector< double > [g_ee](#)
- std::vector< unsigned int > [indInG_ee](#)
- std::vector< unsigned int > [ind_ee](#)
- double * [garray_ei](#)
- std::vector< double > [g_ei](#)
- std::vector< unsigned int > [indInG_ei](#)
- std::vector< unsigned int > [ind_ei](#)
- double * [garray_ie](#)
- std::vector< double > [g_ie](#)
- std::vector< unsigned int > [indInG_ie](#)
- std::vector< unsigned int > [ind_ie](#)
- double * [garray_ii](#)
- std::vector< double > [g_ii](#)
- std::vector< unsigned int > [indInG_ii](#)
- std::vector< unsigned int > [ind_ii](#)

11.94.1 Detailed Description

This file is part of a tool chain for running the Izhikevich network model.

11.94.2 Function Documentation

11.94.2.1 int [main](#) (int *argc*, char * *argv*[])

11.94.2.2 int [printVector](#) (vector< unsigned int > & *v*)

11.94.2.3 int [printVector](#) (vector< double > & *v*)

11.94.3 Variable Documentation

11.94.3.1 `std::vector<double> g_ee`

11.94.3.2 `std::vector<double> g_ei`

11.94.3.3 `std::vector<double> g_ie`

11.94.3.4 `std::vector<double> g_ii`

11.94.3.5 `double* garray`

11.94.3.6 `double* garray_ee`

11.94.3.7 `double* garray_ei`

11.94.3.8 `double* garray_ie`

11.94.3.9 `double* garray_ii`

11.94.3.10 `double gsyn`

11.94.3.11 `unsigned int* ind`

11.94.3.12 `std::vector<unsigned int> ind_ee`

11.94.3.13 `std::vector<unsigned int> ind_ei`

11.94.3.14 `std::vector<unsigned int> ind_ie`

11.94.3.15 `std::vector<unsigned int> ind_ii`

11.94.3.16 `std::vector<unsigned int> indInG_ee`

11.94.3.17 `std::vector<unsigned int> indInG_ei`

11.94.3.18 `std::vector<unsigned int> indInG_ie`

11.94.3.19 `std::vector<unsigned int> indInG_ii`

11.94.3.20 `randomGen R`

11.94.3.21 `randomGen Rind`